UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Valdir Luiz Hofer Arnhold

IMPLEMENTAÇÃO DE UM FRAMEWORK PARA O DESENVOLVIMENTO DE AGENTES COMO SISTEMA MULTI-CONTEXTO

Florianópolis

2018

Valdir Luiz Hofer Arnhold

IMPLEMENTAÇÃO DE UM FRAMEWORK PARA O DESENVOLVIMENTO DE AGENTES COMO SISTEMA MULTI-CONTEXTO

Trabalho de Conclusão de Curso submetido ao Curso de Sistemas de Informação para a obtenção do Grau de Bacharel em Sistemas de Informação.
Orientadora: Prof[a]. Jerusa Marchi, Dr[a] .
Coorientador: Thiago Ângelo Gelaim, MSc.

Florianópolis

2018

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Arnhold, Valdir Luiz Hofer
Implementação de um framework para o desenvolvimento de agentes como sistema multi-contexto / Valdir Luiz Hofer Arnhold ; orientadora, Jerusa Marchi, coorientador, Thiago Ângelo Gelaim, 2018.
108 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Sistema de Informação, Florianópolis, 2018.

Inclui referências.

1. Sistema de Informação. 2. Inteligência Artificial. 3. Agentes. 4. Multi-Contexto. 5. Framework. I. Marchi, Jerusa . II. Gelaim, Thiago Ângelo. III. Universidade Federal de Santa Catarina. Graduação em Sistema de Informação. IV. Título.

Valdir Luiz Hofer Arnhold

IMPLEMENTAÇÃO DE UM FRAMEWORK PARA O DESENVOLVIMENTO DE AGENTES COMO SISTEMA MULTI-CONTEXTO

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Sistemas de Informação", e aprovado em sua forma final pelo Curso de Sistemas de Informação.

Florianópolis, 30 de outubro 2018.

Prof. Cristian Koliver, Dr.
Coordenador do Curso

Prof[a]. Jerusa Marchi, Dr[a] .
Orientadora

Thiago Ângelo Gelaim, MSc.
Coorientador

Banca Examinadora:

Prof. Dr. Ricardo Azambuja Silveira

Prof. Dr. Maicon Rafael Zatelli

Dedico este trabalho ao meu pai José Hugo
e minha mãe Teresinha.

RESUMO

A inteligência artificial (IA) é tema de estudo em diversas áreas de conhecimento. Ao longo da história, ela passou por diferentes etapas e contribui inovando em diferentes frentes. Hoje, a IA é utilizada como suporte para o desenvolvimento de diferentes ferramentas afetando diretamente a vivência e a experiência de uso dos usuários. Uma das frentes da IA se dedica ao estudo e desenvolvimento de modelos e arquiteturas para a criação de agentes cognitivos. Sistemas baseados em agentes, podem ser utilizados em diferentes contextos, fato que ressalta a importância do desenvolvimento de novos métodos a fim de possibilitar sua criação ou prototipação. A literatura, apresenta diversas abordagens e métodos para a criação de modelos de agente entre elas a de Sistemas Multi-Contexto. Este trabalho, apresenta a criação de um *framework* para permitir o desenvolvimento e criação de agentes BDI vistos como um Sistema Multi-Contexto. Como principais resultados, é demostrada a execução de um agente BDI visto como sistema Multi-Contexto e o uso de regras de ponte para raciocinar sobre diferentes contextos.

Palavras-chave: Agentes. Sistema Multi-Contexto. *Framework.*

ABSTRACT

Artificial intelligence (AI) is subject of study in several areas of knowledge. Throughout history, it has gone through different stages and contributed for innovating on different fronts. Today, the AI is used as support for the development of different tools directly affecting the user experience and his way to deal with tecnology. One of the fronts of AI is dedicated to the study and development of models and architectures for the development of cognitive agents. Agent-based systems can be used in different contexts, a fact that emphasizes the importance of the development of new methods in order to enable their creation or prototyping. The literature presents several approaches and methods for the development of agent models including Multi-Context Systems. This work presents the development of a framework to conceive BDI agents as a Multi-Context System. As main results, it is demonstrated the execution of a BDI agent seen as a Multi-Context system and the use of bridge rules to reason about different contexts.

Keywords: Agents. Multi-Context System. Framework

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

Um computador em sua forma original, não possui habilidade para decidir por conta própria o que fazer, cada ação deve estar programada de forma explícita em um programa (RUSSELL; NORVIG, 2009). Em muitos casos, aplicações tradicionais, sem inteligência e raciocínio, satisfazem seus usuários conseguindo executar todas as tarefas propostas. Porém, com o avanço da tecnologia, é exigido cada vez mais que determinadas aplicações possam pensar e decidir por si mesmas (WOOLDRIDGE, 2002, p.14). Esta necessidade não é satisfeita com o uso da computação em sua forma tradicional. Frente a isso, é introduzido o conceito de IA (Inteligência Artificial), um campo de estudo interdisciplinar que busca a resolução de problemas complexos não tradicionais.

A IA faz uso de conhecimentos de diferentes áreas, por exemplo da Biologia, da Antropologia, da Psicologia e da Matemática, para a construção de seus modelos. Devido a essa generalidade, sua pesquisa é dividida em diferentes campos de atuação, que vão do geral ao específico sendo relevante para qualquer tarefa intelectual (RUSSELL; NORVIG, 2009).

Entre os subcampos, uma área bastante ativa na literatura busca estudar, definir e criar ferramentas para o desenvolvimento de agentes artificiais. Um agente, é um sistema computacional inserido em um ambiente e capaz de executar ações autônomas para satisfazer seus objetivos pré-designados (WOOLDRIDGE, 2002). Segundo Wooldridge e Jennings (1995), um agente é composto por uma teoria, uma arquitetura e uma linguagem.

Uma das arquiteturas disponíveis, o modelo *Belief-Desire-Intention* BDI baseado na teoria do raciocínio prático, utiliza um conjunto de crenças, desejos e intenções para representar o conhecimento acerca do mundo, e com base neste conhecimento são realizadas ações no ambiente (RAO; GEORGEFF, 1995, p.315). Além da arquitetura, para modelagem e construção de um agente algumas propriedades devem ser consideradas: a autonomia, a habilidade social, a reatividade e a pró atividade (WOOLDRIDGE; JENNINGS, 1995, p.4). Por outro lado, é importante enfatizar que uso destas propriedades está atrelado ao contexto de uso do agente e consequentemente o problema que se busca resolver com ele.

1.1 PROBLEMATIZAÇÃO

Para (CASALI, 2008) abordagens de sistemas de agentes possuem um grande potencial, porém, ainda há uma série de desafios tecnológicos e teóricos que precisam ser resolvidos, dentre eles, a criação de ferramentas para a modelagem e o desenvolvimento de agentes e a integração de componentes. Sistemas de agentes apresentam dificuldades para integrar diferentes representações de conhecimento, o que muitas vezes impede sua expansão.

De forma mais específica, há um distanciamento entre a formalidade das lógicas com sua devida implementação (HERZIG et al., 2017; CASALI, 2008; SABATER et al., 2000). Além disso, um agente, de forma geral, é bastante restrito. Isso é, as formas para representação de seus estados mentais é limitada para conjuntos de símbolos proposicionais ou suas negações, que como consequência acaba limitando seu uso para determinados contextos.

Tipicamente, as linguagens de programação de agentes são restritas devido a questões relacionadas a eficiência de seu ciclo de raciocínio ou a complexidade das lógicas existentes (HERZIG et al., 2017, p.79). Lógicas de raciocínio podem ser representadas utilizando contextos, que também permitem modelar a generalidade do raciocínio sobre diferentes formas (CASALI, 2008, p.48).

Com essas premissas, a representação modular do agente, através do uso de um Sistema Multi-Contexto, permite representar e isolar lógicas complexas em um determinado contexto, usando regras de ponte para integrar conhecimento entre diferentes contextos (CASALI, 2008; SABATER et al., 2000). O distanciamento entre a teoria e a implementação no desenvolvimento de agentes pode ser encurtado fazendo uso de SMC (Sistemas Multi-Contexto) (SABATER et al., 2000). Um SMC é definido como um *framework* que possibilita a integração de diferentes sistemas formais, por meio de regras de ponte (GIUNCHIGLIA; SERAFINI, 1994).

Em Gelaim et al. (2018) é apresentado um modelo para a definição de agentes vistos como SMC, desenvolvido de forma conjunta com esta pesquisa que teve participação ativa na definição dos principais conceitos definidos. O modelo, permite a definição de um agente genérico que pode ser facilmente adaptado e especializado conforme a necessidade de seu uso. Ele, é originalmente inspirado no trabalho de Gelaim (2016), que é apresentado e detalhado no capítulo 3. O modelo de Gelaim et al. (2018), é utilizado como base e núcleo para a implementação do *framework* proposto por esta pesquisa.

1.2 OBJETIVOS

1.2.1 Objetivo geral

Este trabalho tem por objetivo geral implementar um *framework* que permite o desenvolvimento de agentes BDI sobre um sistema de multi-contexto.

1.2.2 Objetivos Específicos

Buscando realizar o objetivo geral, são definidos os seguintes objetivos específicos:

- Descrever o estado da arte sobre agentes e modelos de agentes como SMC;
- Apresentar o modelo de agente a ser utilizado;
- Apresentar a arquitetura, a gramática e a linguagem de agentes utilizada como base do *framework*;
- Implementar o *framework* com base no modelo definido;
- Permitir a integração do agente com diferentes tipos de ambientes e outros agentes, por meio de sensores e atuadores; e
- Realizar simulações e análise de desempenho e de expressividade com o *framework* implementado.

1.3 ORGANIZAÇÃO DO TRABALHO

No capítulo 2 são apresentados os principais conceitos que fundamentam esta pesquisa. No capítulo 3 são mostrados os trabalhos relacionados com esta pesquisa. A apresentação do modelo e a implementação do *framework* é feita no capítulo 4. As avaliações do *framework* são feitas no capítulo 5. Concluindo, o capítulo 6 mostra as considerações finais e os trabalhos futuros.

2 FUNDAMENTAÇÃO TÉORICA

Este capítulo tem como objetivo definir os conceitos que fundamentam esta pesquisa. Agente, é definido na seção 2.1 dando um destaque para a arquitetura BDI. Na seção 2.2 é definida a entidade ambiente no contexto de um agente. Por fim na seção Na seção 2.3 é descrito um sistema Multi-Contexto, objetivando a sua integração com agentes BDI.

2.1 AGENTE

A autonomia é uma propriedade fundamental para a definição de um agente (WOOLDRIDGE, 2002, p.5). Os atributos e as características que definem um agente tem diferente importância conforme o seu contexto de aplicação, o que causa a falta de um conceito unânime. (WOOLDRIDGE, 2002, p.5).

A propriedade da autonomia está ligada a capacidade de perceber e alterar o ambiente sem a intervenção de outras entidades, ao longo do tempo, a fim de alterar suas percepções no futuro (FRANKLIN; GRAESSER, 1997). Porém, em grande parte das aplicações não existe um domínio completo sobre o ambiente (WOOLDRIDGE, 2002, p.5).

Apesar das diferentes definições, de forma geral, um agente é um sistema computacional que inserido em um ambiente, é capaz de executar ações autônomas para realizar os seus objetivos (WOOLDRIDGE, 2002, p.5). Normalmente um agente, possui um conjunto de ações que pode executar, mas nem todas podem ser sempre executadas, por exemplo, só é possível comprar uma ferrari quando se tem dinheiro suficiente para tal ação (WOOLDRIDGE, 2002, p.5). Um agente pode ser considerado inteligente quando é capaz de executar ações autônomas de forma flexível, sendo que a flexibilidade é baseada na reatividade, na pró-atividade e na habilidade social. (WOOLDRIDGE, 2002, p.5).

Para Wooldridge e Jennings (1995), o desenvolvimento de um agente, deve considerar a sua teoria, a sua arquitetura e a linguagem: A teoria deve definir as formas de conceituar o agente e quais as suas propriedades; A arquitetura deve responder como construir um *software* respeitando as propriedades das teorias; e A linguagem deve permitir desenvolver agentes, com base em uma arquitetura, de forma eficiente. Estendendo o modelo proposto por Wooldridge e Jennings (1995), este trabalho apresenta uma quarta camada denominada *framework* res-

ponsável pela operacionalização do agente fazendo uso da linguagem, da arquitetura e da teoria. Foi utilizada a definição *framework* por se tratar de uma unidade de *sotfware* executável.

Wooldridge (2002) destaca 4 classes concretas de arquiteturas agentes:

1. Baseados em lógica: A tomada de decisões é realizada através de lógica de dedução;

2. Agentes reativos: A tomada de decisão é implementada usando um mapeamento direto da situação para a ação;

3. Agentes *Belief-Desire-Intention*: A tomada de decisão é feita usando uma estrutura de dados para representar crenças (C), desejos (D) e intenções (I) e a relação entre esses estados mentais;

4. Arquitetura em camadas: A tomada de decisão é feita usando camadas de *software*, em diferentes níveis de abstração, para representar o raciocínio sobre o ambiente.

Na seção 2.1.1 será detalhada a arquitetura de agentes BDI, na qual se alicerça o modelo proposto por Gelaim et al. (2018) e que serve como núcleo pré-definido no *framework* proposto neste trabalho.

2.1.1 Agente BDI

A arquitetura de agentes *Belief-Desire-Intention* (BDI), tem suas raízes baseadas na teoria filosófica do raciocínio prático, proposta por Bratman (1987), que tem como princípio, decidir momento a momento qual ação realizar, para alcançar os objetivos (WOOLDRIDGE, 2002, p28). O raciocínio prático é composto por dois importantes processos: a deliberação que busca decidir quais objetivos se deseja alcançar; e o raciocínio meio-e-fim que busca responder como alcançar estes objetivos (WOOLDRIDGE, 2002, p28).

Crenças representam o conhecimento sobre o mundo, desejos o estado do mundo que o agente deseja alcançar e intenções representam desejos que o agente de fato vai tentar realizar (WOOLDRIDGE, 2002). Para Wooldridge (2002) as intenções possuem um papel importante no processo do raciocínio prático, e suas propriedades são descritas da seguinte forma:

1. Intenções governam o raciocínio meio-e-fim: se um agente possuir uma intenção, então vai tentar realizá-la, para isso, ele pre-

cisa planejar como a realizar; caso não consiga por um caminho planejado ele deve tentar por outros.

2. Intenções restringem a deliberação futura: se um agente possui uma intenção ele não vai se comprometer com outras intenções que negam ou restringem sua intenção atual.

3. Intenções persistem: se um agente possui uma intenção ele vai persistir para a realizar até acreditar que ela foi realizada, a não ser que ele passe a acreditar que não é mais possível realizá-la ou sua motivação não é mais válida.

4. Intenções influenciam as crenças sobre as quais raciocínio prático futuro é baseado: com base em sua intenção, o agente pode planejar futuras ações, acreditando que realizará a intenção.

Existem 3 tipos de relação entre os estados mentais de um agente BDI, denominados realismos, definidos por Rao e Georgeff (1991), a definição de cada um dos conjuntos é descrita da seguinte forma:

1. realismo forte: o conjunto de intenções é um subconjunto de desejos que, por sua vez, é um subconjunto das crenças. Isso é, se um agente não acredita em uma propriedade, não a desejará e nem pretenderá;

2. realismo: o conjunto de crenças é um subconjunto de desejos que, por sua vez, é um subconjunto do conjunto de intenções. Isso é, se um agente acredita que algo é possível, ele o desejará e pretenderá; e

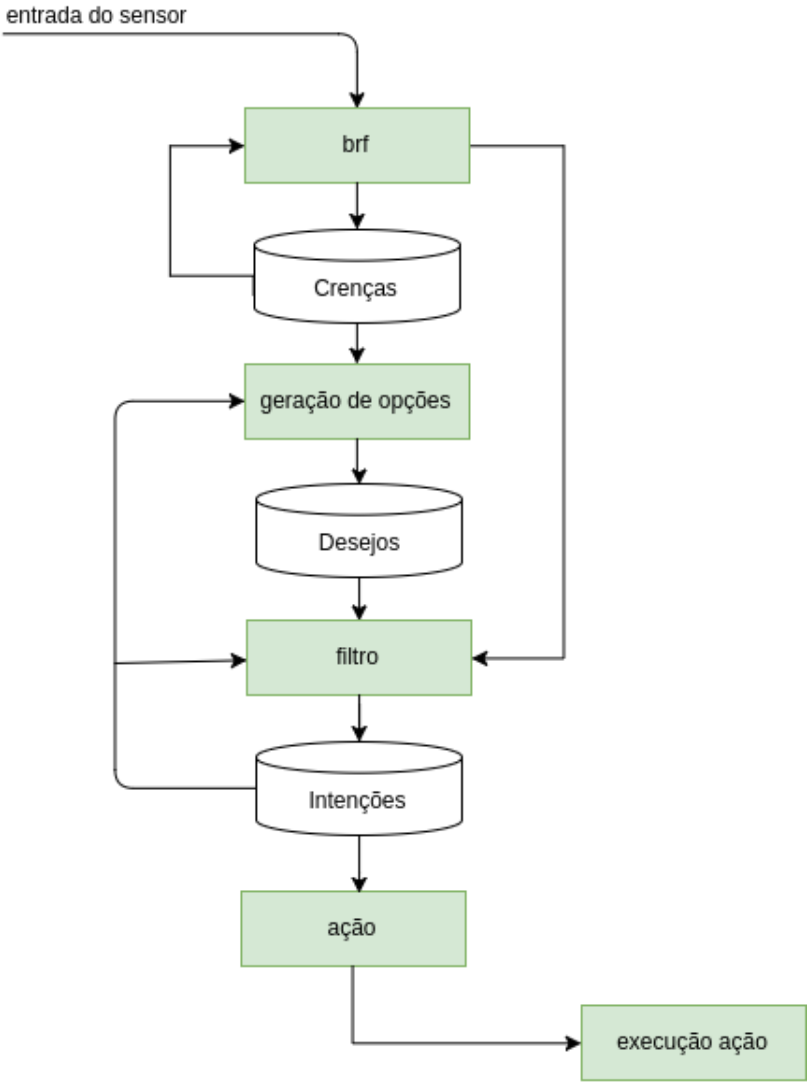
3. realismo fraco: o agente não deseja uma propriedade se acredita em sua negação; não possui intenção de propriedade se sua negação é desejada e não possui intenção de uma propriedade se acredita em sua negação .

O diagrama de uma arquitetura BDI genérica proposto por Wooldridge (2002), é mostrado na Figura 1. Retângulos representam funções e cilindros estados mentais. A especificação de cada um dos componentes é feita seguinte forma:

- A função *brf* representa a revisão de crenças, que determina um novo conjunto de crenças;

- a função de geração de opção é responsável por determinar os desejos do agente com base em suas crenças sobre o mundo e suas intenções;

Figura 1 – Diagrama esquemático de uma arquitetura BDI genérica



Fonte: (WOOLDRIDGE, 2002, p31).

- a função de filtro, representa a deliberação do agente e determina as intenções do agente com base em suas crenças, desejos e

intenções;

- as intenções representam o foco atual do agente, são estados com os quais ele está comprometido;
- a função de seleção de ação determina qual ação executar com base em suas intenções atuais (WOOLDRIDGE, 2002).

Com base na arquitetura mostrada, o ciclo de raciocínio simplificado de um agente BDI, proposto por Wooldridge (2000), é mostrado no Algoritmo 1:

Algoritmo 1: Ciclo de raciocínio BDI

```
1 while true do
2 |   observe o mundo;
3 |   atualize o modelo de mundo interno;
4 |   delibere sobre a próxima intenção que deseja alcançar;
5 |   utilize raciocínio meio-e-fim para obter um plano para a
  |     intenção;
6 |   execute o plano;
7 end
```

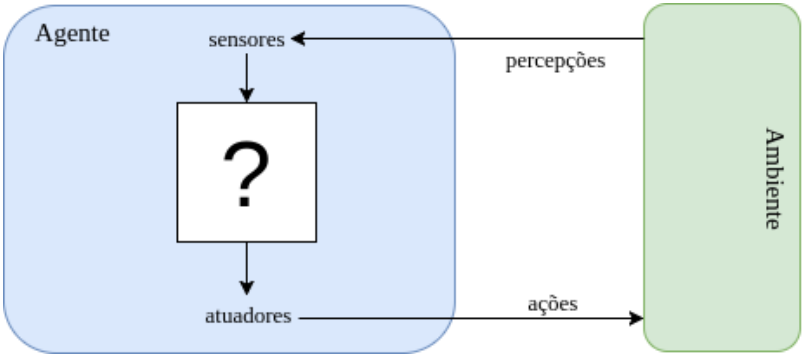
2.2 AMBIENTE

Conforme foi mostrado na seção 2.1, um agente interage com seu ambiente. Essa interação é feita através de sensores que capturam percepções e atuadores que agem no mundo (RUSSELL; NORVIG, 2009). Este conceito é ilustrado na Figura 2.

Segundo Russell e Norvig (2009) um ambiente possui as seguintes propriedades:

- Completamentamente observável ou parcialmente observável: Um ambiente é completamente observável se os sensores do agente são capazes de dar acesso ao estado completo do mundo em todos os instantes do tempo. Caso haja ruídos nos sensores ou parte do estado não possui visibilidade, o ambiente é parcialmente observável.
- Determinista ou estocástico: Se o próximo estado do ambiente for determinado de forma total pelo estado atual e pela ação do

Figura 2 – Interação de um agente com o ambiente através de sensores e atuadores



Fonte: (RUSSELL; NORVIG, 2009, p35).

agente então o ambiente é considerado determinista, caso contrário é estocástico.

- Episódico ou sequencial: Em um ambiente episódico a experiência do agente é segmentada em episódios atômicos, isso é, em um episódio o agente recebe uma percepção e executa uma única ação. Por outro lado, em ambientes sequencias uma ação pode interferir em ações futuras não sendo atômica, ambiente sequencias possuem um maior custo de gerenciamento.

- Estático ou dinâmico: Um ambiente estático se ele não altera seu estado durante o processo de deliberação do agente, caso contrário ele é dinâmico.

- Discreto ou contínuo: A distinção entre um ambiente discreto ou contínuo é feita a partir dos estados do ambiente, o tempo, e as percepções e ações do agente. Por exemplo, um ambiente de xadrez que possui um conjunto discreto de estados e de percepções e ações, por outro lado, é contínuo em relação ao tempo.

- Conhecido ou desconhecido: Um ambiente é conhecido se o agente possui sabedoria das regras que regem o ambiente, por exemplo um jogo de cartas, caso contrário ele é desconhecido. Um ambiente pode ser completamente observável e desconhecido.

Feita a definição de um agente e do seu ambiente a seção seguinte mostra uma das abordagens para a construção de agentes, utili-

zado como base nesta pesquisa devido ao poder de representatividade e integração de diferentes estados mentais.

2.3 SISTEMAS MULTI-CONTEXTO

Um SMC é definido como um modelo que permite a integração de diferentes sistemas formais (GIUNCHIGLIA; SERAFINI, 1994). A especificação de um SMC é feita por uma tripla, composta por: unidades ou contextos, lógicas e regras de ponte (CASALI, 2008).

A noção de contexto é estudada em diferentes áreas de conhecimento, e de forma especial na Inteligência Artificial. Um contexto é uma ferramenta para formalizar a localidade do raciocínio ou para resolver o problema da generalidade (CASALI, 2008, p48). O uso de contextos, deve seguir dois princípios definidos em Ghidini e Giunchiglia (2001):

1. Princípio da localidade: O processo de raciocínio utiliza apenas parte do potencial disponível (por exemplo, os processos de inferência disponíveis), esta parte é denominada contexto.

2. Princípio da compatibilidade: Existe uma compatibilidade entre os raciocínios realizados em diferentes contextos.

O princípio da localidade permite que uma mesma teoria tenha uma semântica diferente, conforme o contexto em que está inserida, e essa é uma importante propriedade para a construção de agentes como um SMC (GHIDINI; GIUNCHIGLIA, 2001).

As regras de ponte relacionam formulas de diferentes contextos, e cada regra pode ser entendida como uma regra de inferência com premissas e conclusões em diferentes contextos (CASALI, 2008). A propriedade de integração, dada pelas regras de ponte, de diferentes lógicas, que podem ser complexas, é citada como uma das vantagens do uso de SMC por Casali (2008). Além disso, as regras pontes garantem a aplicação do princípio da compatibilidade, pois elas permitem que a inferência seja feita de forma mútua, uma regra de ponte altera a teoria de seu contexto usando derivação de fórmulas de outros contextos (CASALI, 2008, p48).

Definição 2.3.1. Para Casali (2008, p48), um SMC é definido como um grupo de unidades interligadas:

$$\langle \{C_i\}\, i \in I, \Delta_{rp} \rangle$$

onde $\{C_i\}$ é um conjunto de contextos, Δ_{rp} são as regras de ponte, que relacionam fórmulas em contextos diferentes e I é uma lista enumerada de índices. Para cada $i \in I$, C_i é um sistema formal axiomático:

$$\langle L_i, A_i, \Delta_i \rangle$$

sendo L_i uma linguagem, A_i os axiomas e Δ_i as regras de inferência.

Feita a definição de um SMC a seção seguinte mostra como é feita a modelagem de um agente como um SMC.

2.3.1 Agentes BDI como Sistema Multi-Contexto

Conforme foi mostrado na seção 2.1.1 um agente BDI é composto por representações de crenças, desejos e intenções. Utilizar uma estrutura lógica unificada para fazer a modelagem de um agente BDI pode ser muito complexo, além disso, gerenciar diferentes fórmulas em um único núcleo também adiciona complexidade a modelagem (CASALI, 2008, p50). Frente a isso, a literatura apresenta vários modelos de agentes como SMC baseados em BDI. Por exemplo, um agente BDI como SMC pode ter associado a ele contextos para representar noções intencionais, teorias de crenças, desejos e intenções, onde cada contexto possui uma lógica adequada associada a ele, e a integração entre os contextos é feita por meio de regras de ponte (CASALI, 2008).

O uso de SMC para a representação de agentes BDI permite representar diferentes estados mentais em diferentes contextos, o que agrega poder representacional se comparado a arquiteturas que utilizam uma única lógica ou várias lógicas em uma estrutura global (CASALI, 2008, p50). Esse poder de expressividade é, por exemplo, mostrado no trabalho de Gelaim (2016) que estende um agente BDI para suportar confiança por meio de um SMC.

Além disso, para Sabater et al. (2000) a abordagem de SMC fornece um caminho entre a especificação e a implementação de agentes, preenchendo essa lacuna que existe entre a teoria e a prática. Sob o ponto de vista da engenharia de *software* o uso de SMC suporta a decomposição modular e o encapsulamento seguindo as boas práticas de desenvolvimento de software. Para corporificar a definição de um agente visto como SMC, no capítulo 3 é descrito um modelo de agentes baseado em um SMC.

2.4 CONSIDERAÇÕES FINAIS

Este capítulo mostrou os principais conceitos que definem um agente, detalhando suas propriedades e relação com o ambiente por meio de sensores e atuadores. Apresentou um modelo de desenvolvimento de agentes composto pela definição da teoria, da arquitetura e da linguagem que foi utilizado como base para o método desta pesquisa. Foi dado um foco especial para arquitetura BDI, utilizada como base para o *framework* implementado por esta pesquisa. Por fim, foi definido um SMC que fundamenta o modelo apresentado em Gelaim et al. (2018) que define a construção do *framework* deste trabalho.

3 TRABALHOS CORRELATOS

Neste capítulo, são apresentados os principais trabalhos, que de alguma forma, se relacionam à proposta apresentada nesta pesquisa. São mostrados trabalhos com abordagens de agentes como Sistema Multi-Contexto e trabalhos que apresentam a definição de um *framework* para o desenvolvimento de agentes. Em particular, detalha-se o modelo proposto em Gelaim (2016), usado como base e inspiração para o modelo proposto em Gelaim et al. (2018) que, por sua vez, fundamenta o *framework* proposto nesta pesquisa.

3.1 MODELO DE AGENTES E-BDI INTEGRANDO CONFIANÇA BASEADO EM SISTEMAS MULTI-CONTEXTO

Gelaim (2016) apresenta um modelo de agente inspirado em BDI que integra um modelo computacional de emoções. A abordagem utilizada para o modelo é baseada em um Sistema Multi-Contexto. No modelo proposto, as crenças são graduadas com graus de certeza do agente sobre elas e podem ser rotuladas como especiais. As crenças especiais são chamadas de julgamentos e são utilizadas para intermediar a relação entre confiança e emoções.

O modelo de agente proposto por Gelaim (2016) é mostrado na formalmente na Definição 1.

Definição 1 (Modelo de agente proposto)**.**

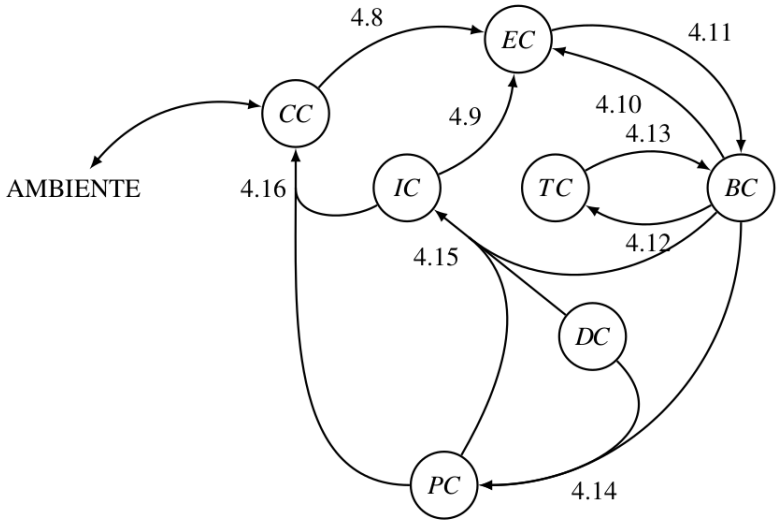
$$AG = \langle \{BC, DC, IC, PC, TC, EC, CC\}, \Delta_{rp} \rangle \tag{3.1}$$

onde BC, DC, IC, PC, TC, EC, CC são, respectivamente, os contextos de crenças, desejos, intenções, planejamento, confiança, emoções e comunicação; e Δ_{rp} *são as regras de ponte utilizadas para trocar informações entre contextos (*GELAIM*, 2016).*

Considerando que o modelo baseia-se em um sistema Multi-Contexto, a integração entre os contextos é feita por meio das regras de ponte, conforme mostra a Figura 3. Nesta figura, podem ser observados os contextos definidos na Definição 1 e o conjunto de regras de ponte, enumeradas, conforme a seção onde são detalhadas no trabalho de (GELAIM, 2016).

O contexto de comunicação (CC) representa uma interface entre os contextos internos do agente e seu ambiente sendo responsável pelas

Figura 3 – Interação entre contextos por meio de regras de ponte



Fonte: (GELAIM, 2016)

percepções. Para realizar raciocínio, este contexto utiliza lógica de primeira ordem restrita a cláusulas de Horn. As percepções recebidas do ambiente são classificadas em duas categorias:

1. Mensagens: recebidas de outros agentes, representado pelo predicado *receive message*; e

2. Observações: feitas no ambiente, representado pelo predicado *observe*.

Para executar ações no ambiente é definido, no contexto de comunicação, um predicado especial chamado *does*. Os três predicados especiais, *receive message*, *observe* e *does*, são utilizados pelo contexto de emoções para gerar estímulos emocionais.

O contexto de emoções (EC) é responsável por manipular e representar as emoções do agente. Considerando que a literatura na área de emoções apresenta diversos modelos, o trabalho de (GELAIM, 2016) optou em utilizar o WASABI (BECKER-ASANO, 2008) para integrar fatores emocionais às crenças. O critério para a escolha deste modelo foi a flexibilidade de integração com *frameworks* de agentes cognitivos. O

objetivo do contexto de emoções é permitir ao agente uma experiência afetiva.

O contexto de crenças (BC) representa o conhecimento do agente sobre o mundo. As crenças, podem possuir gradações que representam o grau de certeza de sua credibilidade. Para sua representação, são definidos dois predicados especiais:

1. J: que representa os julgamentos; e

2. C: que representa crenças sobre a confiança.

A linguagem de crenças no trabalho de (GELAIM, 2016) é baseada no modelo definido por (CASALI; GODO; SIERRA, 2005) sendo estendida para contemplar julgamentos e confiança.

O contexto de confiança (TC) é responsável por determinar a confiança do agente em outras entidades. Para isso são utilizadas informações do contexto de crenças que são integradas, por meio de regras de ponte, ao contexto de confiança.

O contexto de desejos (DC) representa os objetivos gerais do agente. Os desejos são classificados como positivos ou negativos. O agente espera realizar os desejos positivos e não concretizar os desejos negativos. Assim como as crenças os desejos também são graduados, isso é, o agente pode desejar mais que um determinado objetivo se concretize em detrimento a outros. A linguagem para representação dos desejos, em (GELAIM, 2016), é baseada no trabalho de (CASALI; GODO; SIERRA, 2005).

O contexto de intenções (IC) representa as intenções do agente que são geradas com base em suas crenças e desejos. As intenções são graduadas objetivando avaliar os custos e benefícios envolvidos em ações. A linguagem utilizada por este contexto é definida em (CASALI; GODO; SIERRA, 2005).

O contexto de planejamento (PC) é responsável por sintetizar que planos que satisfazem os desejos do agente. Assim como no contexto de comunicação, a linguagem utilizada é baseada em lógica de primeira ordem restrita a cláusulas de Horn. Além disso são definidos três predicados especiais: *action*, *plan* e *best plan*.

Fazendo um comparativo entre o trabalho de (GELAIM, 2016) e (GELAIM et al., 2018), ambos são baseados em um modelo de Sistema Multi-Contexto. Em (GELAIM et al., 2018) o trabalho original foi estendido e generalizado para suportar a criação de novos contextos. Essa característica permite que sejam criadas novas arquiteturas de agentes usando como núcleo o modelo proposto por (GELAIM et al., 2018).

Além disso, em (GELAIM, 2016) a definição do modelo preza por sua formalidade em detrimento da especificação técnica para uma possível implementação. Já em (GELAIM et al., 2018) são mostrados detalhes técnicos que permitem a a implementação de um agente como SMC. Esta especificação técnica é implementada no *framework* proposto por esta pesquisa.

3.2 BDI AGENT PROGRAMMING IN AGENTSPEAK USING JASON

O trabalho de Bordini e Hübner (2006) tem como objetivo mostrar as principais funções disponíveis no *Jason* que é um framework para o desenvolvimento de agentes BDI baseado em *AgentSpeak* que é uma linguagem de programação orientada a agentes baseada em programação lógica. A arquitetura do AgentSpeak é inspirada em BDI, e em sua versão original é considerada uma linguagem abstrata. Esta característica se assemelha a este trabalho, considerando que os modelos de agente como sistema Multi-Contexto são restritos a teorias e arquiteturas abstratas. Frente a isso, Bordini e Hübner (2006) adicionam ao *AgentSpeak* extensões que são necessárias para a execução e operacionalização de agentes.

Jason[1] é implementado em Java, assim como o *framework* proposto neste trabalho, as principais funcionalidades disponíveis são: Comunicação entre agentes; anotações em crenças e em planos; execução de sistema multi-agente sobre uma rede; customização em Java das funções do agente; possibilita a implementação de ambiente multi-agente.

Considerando que sua implementação é baseada em *AgentSpeak*, o agente é definido com um conjunto de crenças que determinam o estado inicial da base de crenças. A linguagem de crenças é definida como um conjunto de fórmulas atômicas criadas sobre lógica de primeira ordem. Além disso é definido um conjunto de planos que podem satisfazer objetivos. Os objetivos são classificados como *achievement goals* e *test goals*.

Os planos são implementados por meio de um sistema reativo. Isso é eles reagem à mudanças que acontecem na base de crenças do agente. Por exemplo, ao receber uma nova percepção do ambiente ela é adicionado pelo agente em sua base de crenças disparando um evento que por consequência irá executar um plano. Os planos são definidos pelo programador do agente.

[1] http://jason.sourceforge.net

A comunicação do agente no Jason é inspirada em KQML, que define as seguinte funções: *tell*; *untell*; *achieve*; *unachieve*; *tellHow*; *untellHow*; *askIf*; *askAll*; *askAll* e *askHow*.

A troca de mensagens é feita de forma assíncrona. As mensagens são internamente armazenadas em uma caixa de entrada sendo processados pelo agente a cada ciclo de raciocínio. O início do ciclo de raciocínio é determinado por uma função de seleção da mensagem, que pode ser personalizado pelo programador do agente.

Uma das funcionalidades do Jason é a criação de ambiente. Em Jason, um agente pode atuar em ambientes reais ou virtuais. Para facilitar a criação de protótipos é fornecida uma classe *Environment* que pode ser estendida pelo programador para a criação de seus próprios ambientes. Diferente do modelo de agente proposto por (GELAIM et al., 2018), em (BORDINI; HÜBNER, 2006) os sensores e atuadores ficam atrelados ao ambiente no qual o agente está inserido, essa característica dificulta a integração de agentes *Jason* com ambientes de diferentes topologias.

Um agente *Jason* é representado pela classe *Agent*. Para customizar funções do agente em *Jason* o programador deve sobrescrever métodos da classe *Agent*. Por exemplo para modificar a seleção das percepções deve ser sobrescrito o método que seleciona uma percepção.

Na arquitetura proposta por (GELAIM et al., 2018), a alteração de características do agente é muito mais simples. Para, por exemplo, alterar a função de seleção de percepções basta modificar a regra de ponte responsável por esse procedimento.

De forma geral, o trabalho de (BORDINI; HÜBNER, 2006) apresenta uma solução completa para o desenvolvimento de agentes BDI. No modelo é definido um ciclo de de raciocínio e a integração entre bases de conhecimento. Além disso são fornecidos diferentes artefatos que buscam facilitar o desenvolvimento de agentes, como por exemplo a criação de ambientes e uma IDE para o desenvolvimento. Por outro lado, a customização de um agente *Jason* considerando sua arquitetura pode ser extremamente custosa.

Comparando o *Jason* ao framework proposto neste trabalho é importante enfatizar que ambos são implementados utilizando como base a *Java Virtual Machine*. *Jason* por ser uma linguagem já consolidada fornece uma solução completa para o desenvolvimento de agentes, como ferramentas de debug, plugins para IDEs, manuais, entre outros, diferente deste trabalho que só implementa uma linguagem. Além disso, a plataforma do *Jason* oferece suporte para sistemas multi agente, diferente deste trabalho que foi projetado para a atuação de um único

agente.

Por outro lado, devido a estruturação do Jason a customização de agentes é mais custosa. Por exemplo, na versão original do *Jason*, sensores e atuadores ficam acoplados ao ambiente. Esta característica facilita o desenvolvimento de agente, mas também dificulta a integração com diferentes tipos de ambiente. Neste trabalho, sensores e atuadores estão ligados ao agente, não sendo definida uma interface para ambientes, o que facilita a integração do agente com outras entidades.

3.3 A PROGRAMMING LANGUAGE FOR COGNITIVE AGENTS GOAL DIRECTED 3APL

O trabalho de Dastani et al. (2004) apresenta a especificação para uma linguagem de agentes cognitivos. A linguagem é baseada na 3APL (An Abstract Agent Programming Language) e permite ao programador a definição de estados mentais como crenças, objetivos, planos e ações.

As crenças de um agente 3APL descrevem o conhecimento do agente sobre o mundo bem como seu conhecimento acerca do seu estado interno. Elas são especificados e armazenadas na base de crenças do agente.

Os objetivos descrevem os estados que o agente deseja alcançar e são especificados pela base de metas do agente. A linguagem de crenças e de objetivos é definida por termos e fórmulas como lógica de primeira ordem. Os termos representam os objetos de domínio e suas fórmulas as relações entre os objetos.

Os planos são responsáveis em atingir os objetivos. Um plano é construído a partir de ações básicas e testes na base de crenças. Planos abstratos são chamados de metas de realização. Uma ação básica especifica os recursos com os quais um agente deve alcançar um determinado estado. O efeito da execução de uma ação básica altera a base de crenças do agente e não o mundo. Por outro lado, ação de teste verifica se uma determinada fórmula é derivável na base de crenças.

Segundo a especificação, um plano abstrato não pode ser executado, porém ele pode ser derivado em ações básicas por meio de raciocínio e regras, o que que permite a sua execução de forma indireta.

Implementar um agente em 3APL significa especificar suas crenças, metas e planos iniciais. Além disso é necessário escrever um conjuntos de regras de revisão de metas, planejar regras de seleção e regras

de revisão de planos. A execução é definida por meio de um ciclo de deliberação, que mostra como as regras são aplicadas e em qual ordem devem ser executadas.

Por fim, o trabalho de Dastani et al. (2004) mostra uma especificação para uma linguagem de agentes, baseada em crenças, objetivos e ações. No trabalho, é definida toda a relação sintática e semântica dos componentes envolvidos. Porém a definição do ciclo de deliberação, que não faz parte do escopo do trabalho, é fraca. Este fator dificulta a implementação da linguagem e consequentemente a operacionalização de agentes.

Comparado a este trabalho, a definição dos planos mostrada em Dastani et al. (2004) é semelhante, considerando que os mesmos são derivados em ações nos dois modelos. Mas, por outro lado, em *3APL* um plano executado altera diretamente a base de crenças. No *framework* implementado neste trabalho, que segue o modelo proposto em Gelaim et al. (2018), essa alteração é feita diretamente no ambiente, devendo ser observada pelo agente para gerar um novo ciclo de deliberação. Um outro fator que é importante comparar entre os dois trabalhos são definições e configurações necessárias para a execução de um agente. Em *3APL* é preciso definir um conjunto de regras, além dos planos e metas. No modelo apresentado no capítulo 4, a definição de regras não é necessária, considerando que, por padrão, é definido um conjunto de regras baseadas em BDI.

3.4 ANÁLISE DOS TRABALHOS RELACIONADOS

O trabalho de Gelaim (2016) apresenta um modelo de agente BDI baseado em Sistema Multi-Contexto, porém não especifica como deve ser sua implementação. Entretanto, é o precursor do trabalho que vem sendo desenvolvido por (GELAIM et al., 2018), que estabelece como deve ser feita a implementação de agentes vistos como SMC e fundamenta a proposta de implementação desta pesquisa, mostrada no Capítulo 4.

O trabalho de Bordini e Hübner (2006) mostra uma solução completa para o desenvolvimento de agentes, deste a especificação até a implementação, parte da arquitetura de software definida em (BORDINI; HÜBNER, 2006) foi utilizada como base para a implementação desta pesquisa. Por outro lado, a integração de um agente JASON com diferentes tipos de ambientes é dificultada devido sua arquitetura. Frente a isto, neste pesquisa foi desenvolvido um modelo de sensores e atuadores

desacoplados da representação do ambiente o que acaba facilitando o uso do agente em diferentes contextos.

O trabalho de Dastani et al. (2004) mostra a especificação de uma linguagem de agentes. Assim como o *framework* implementado nesta pesquisa, o modelo é genérico e permite criar novas configurações para a execução de um agente. Por outro lado, os detalhes técnicos que, de fato, permitem a execução de um agente são limitados. Esta característica acaba dificultando a implementação da arquitetura e do modelo proposto por Dastani et al. (2004).

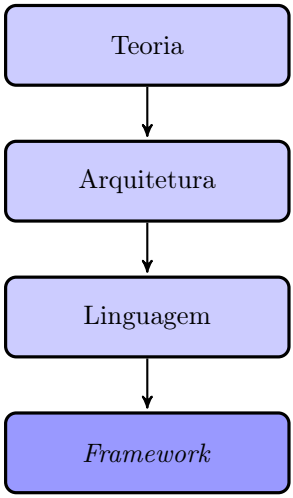
Entre os três trabalhos apresentados, a proposta de Bordini e Hübner (2006) é mais próxima ao *framework* implementado neste trabalho, por tratar de questões relacionadas a implementação de um agente. Além disso, também é desenvolvida em JAVA e implementa uma parser com base em uma linguagem de agente. Relacionado ao modelo de agentes utilizado neste trabalho o trabalho de Gelaim (2016) se assemelha por também definir um agente visto como um SMC.

4 PROPOSTA

Conforme foi abordado no Capítulo 1, existem diversos desafios, tecnológicos e teóricos, atrelados aos modelos de agentes inteligentes. Frente a isso, esta pesquisa propõe a implementação de um *framework* para permitir a operacionalização de agentes como SMC.

O método de desenvolvimento de agentes é adaptado de Wooldridge e Jennings (1995) que, em seu formato original, divide o processo de desenvolvimento em 3 fases: Teoria; Arquitetura; e Linguagem. Como adaptação ao método, é proposta uma quarta fase denominada *Framework* conforme é mostrado na figura 4.

Figura 4 – Método de desenvolvimento de agentes adaptado de Wooldridge e Jennings (1995).



Conforme mostrado na seção de escopo definida no capítulo 1, a teoria, a arquitetura e a linguagem, elementos necessários para a implementação do *framework*, sendo esse o principal objetivo deste trabalho, são definidos na tese *Consciência situacional em agentes autônomos: Raciocínio sobre percepções* de autoria de Thiago Ângelo Gelaim e no artigo *Sigon: A Multi-Context System Language for Intelligent Agents* de Gelaim et al. (2018) desenvolvido de forma conjunta. Por não ser o objetivo central deste trabalho, os aspectos relativos a teoria, arquitetura e linguagem são apenas superficialmente descritos visando apenas

os conceitos utilizados no desenvolvimento do *framework*.

4.1 TEORIA

Considerando que o trabalho apresentado em (GELAIM et al., 2018) se trata de uma arquitetura genérica, que permite a definição de novos contextos e regras de ponte, é possível representar através dela diferentes tipos de teorias. Por exemplo, a criação de um simples agente reativo ou de uma arquitetura mais complexa envolvendo emoções conforme é mostrado nos trabalhos de Gelaim, Silveira e Marchi (2015), SILVEIRA et al. (2016).

Com o objetivo de corporificar o modelo genérico, este trabalho utiliza a teoria do raciocínio prático, por meio de uma arquitetura BDI. Na seção 4.2 será criado um cenário para a aplicação da arquitetura.

4.2 ARQUITETURA

A arquitetura do agente define suas lógicas, seus contextos e as regras de ponte. O modelo genérico do agente, conforme definido em (GELAIM et al., 2018), é mostrado na Definição 2.

Definição 2 (Modelo de agente proposto)**.**

$$AG = \langle CC \cup \bigcup_{i=1}^{n} C_i, \Delta_{br} \rangle \tag{4.1}$$

onde CC é o contexto de comunicação e C_i com $1 \leq i \leq n$ os demais contextos do agente e Δ_{br} é o conjunto de regras de ponte; Uma regra br_i conecta dois ou mais contextos (GELAIM et al., 2018).

Com base nesse modelo, é definido em (GELAIM et al., 2018) uma arquitetura de agente BDI, conforme a Definição 3.

Definição 3 (Definição de um agente BDI)**.**

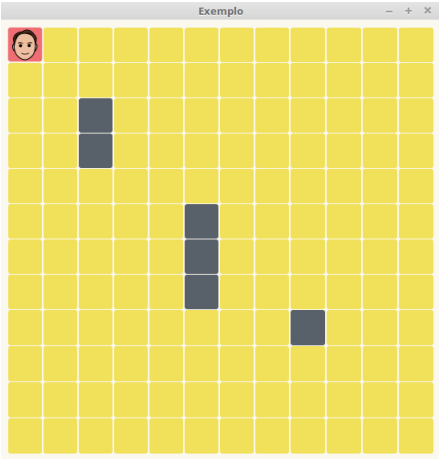
$$AG_{BDI} = \langle \{CC, BC, DC, IC, PC\}, \Delta_{br} \rangle \tag{4.2}$$

onde CC, BC, DC, IC, PC são, respectivamente, contextos de comunicação, crenças, desejos, intenções e planejamento, e Δ_{br} é o conjunto de regras de ponte (GELAIM et al., 2018).

Para exemplificar o uso da arquitetura BDI, esse trabalho pro-

põe um cenário no qual um agente é responsável em se deslocar em um ambiente que pode conter obstáculos. Como regra, o agente deve desviar dos obstáculos que encontrar. Para este cenário o ambiente é representado por uma matriz de duas dimensões, e conforme mostrado na seção 2.2, é parcialmente observável, determinista, episódico, estático e conhecido. A Figura 5 mostra o ambiente, onde as células cinzas representam os obstáculos e as células amarelas o caminho livre. Apesar da simplicidade, essa modelagem envolve o uso de todos os contextos mostrados na Definição 3, além dos elementos: lógicas; regras de ponte; gramática; sensores; e atuadores, que serão especificados nas próximas seções.

Figura 5 – Representação do ambiente



4.2.1 Lógicas

Seguindo a definição de (GELAIM et al., 2018), o modelo proposto é capaz de executar raciocínio sobre lógica proposicional e lógica de primeira ordem restrita a cláusulas de Horn. Utilizando como exemplo o cenário proposto, se utiliza lógica de primeira ordem para, por exemplo, representar a posição atual do agente, da seguinte forma:

posicao(0,0)

O uso de lógica proposicional é feito no exemplo do cenário para representação de obstáculos:

obstaculo
A definição das lógicas do agente é um importante requisito para a construção do agente como um *software*, pois deve ser considerada uma arquitetura que permita raciocinar sobre os tipos de lógica definidos no modelo.

4.2.2 Contextos

Os contextos do agente, seguindo a especificação de Gelaim et al. (2018), são classificados em duas categorias: de estado mental e funcionais. Contextos de estado mental podem ser definidos, por meio da linguagem, pelo desenvolvedor do agente. O conjunto de contextos funcionais é restrito ao contexto de planejamento (PC) e comunicação (CC), permitindo ao desenvolvedor somente a criação de contextos de estado mental.

Considerando um agente BDI, o conjunto de contextos de estados mentais é composto por crenças (BC), desejos (DC), intenções (IC). Para auxiliar e otimizar o desenvolvimento ou a prototipação de agentes, este trabalho implementa esses contextos por padrão no *framework*.

Frente ao cenário proposto, o contexto de crenças contém a posição atual do agente e se existe um obstáculo no caminho. O desejo do agente é chegar em sua posição final. As intenções representam a interface entre os desejos e os planos, sendo assim o agente intenciona chegar em sua posição final e para isso desviar dos obstáculos.

O contexto de planejamento é responsável pelo gerenciamento e construção de planos para satisfazer as intenções do agente, sua formalização, conforme (GELAIM et al., 2018), é:

Definição 4 (Contexto de Planejamento)**.** *Seja PC o contexto de planejamento. PC é definido como:*

$$PC = \{\bigcup_{i=1}^{n} P_i, \bigcup_{j=1}^{m} A_j\}, \tag{4.3}$$

onde P_i *com* $1 \leq i \leq n$ *são os planos do agente e* A_j *com* $1 \leq j \leq m$ *são as ações do agente. Um plano é definido como:*

$$plan(\varphi, \beta,\ Pre,\ Post, c_a) \tag{4.4}$$

onde φ *é o objetivo do plano,* β *é o conjunto de ações, Pre são as pré*

condições, Post são as pós condições e c_a é o custo. Apesar da definição formal do custo de execução do plano, definida por um valor real em (GELAIM et al., 2018), este trabalho não aplicará semântica sobre essa propriedade. Uma ação é definida como:

$$action(\alpha,\ Pre,\ Post, c_a) \tag{4.5}$$

onde α é a ação, Pre são pré condições para a execução de α e Post são pós condições para a execução de α (GELAIM et al., 2018).

No cenário utilizado nessa proposta, o agente deve conter dois planos:

1. Para a locomoção no ambiente: onde o objetivo é chegar em sua posição final, o conjunto de ações contém uma única ação para a locomoção, a pré condição é não estar em sua posição final e a pós condição é estar em sua posição final.

2. Para desviar dos obstáculos: onde o objetivo é desviar do obstáculo, o conjunto de ações contém uma única ação para realizar o desvio, a pré condição é estar frente a um obstáculo e a pós condição é estar livre do obstáculo.

O contexto de comunicação, representa a interface do agente com seu ambiente. Definido em (GELAIM et al., 2018), ele é composto por um conjunto de sensores e atuadores, coforme mostra a Definição 5.

Definição 5 (Contexto de comunicação)**.** *Seja CC o contexto de comunicação. CC é definido como:*

$$CC = \langle \bigcup_{i=1}^{n} S_i \cup \bigcup_{j=1}^{m} A_j \rangle, \tag{4.6}$$

onde S_i com $1 \leq i \leq n$ é o conjunto de sensores do agente, e A_j com $1 \leq j \leq m$ é o conjunto de atuadores do agente. Um sensor é um par ordenado (ω, χ), onde ω representa a identificação do sensor, e χ é a função que recebe a percepção. Um atuador é um par ordenado (ρ, α), onde ρ representa a identificação do atuador, e α é ação que será executada (GELAIM et al., 2018).

A definição dos contextos funcionais do agente é um importante requisito para o desenvolvimento do *framework*, considerando que devem ser implementados sensores, atuadores e uma estrutura de dados para representar os planos.

4.2.3 Regras de ponte

As regras de ponte, são responsáveis pela troca de conhecimento entre os contextos do SMC. Para facilitar o desenvolvimento é proposto em Gelaim et al. (2018) um conjunto de regras que representam o comportamento BDI. Essas regras são implementadas por este trabalho com o objetivo de permitir a execução do cenário proposto.

Para permitir o entendimento do conjunto de regras de ponte utilizados nesse trabalho e considerando um cenário BDI, o Exemplo 1 mostra uma simples regra de ponte envolvendo os contextos de crenças, desejos e intenções. Neste exemplo, caso o agente deseja (DC) ψ e não acredita (BC) em ψ ele passa a ter a intenção (IC) de ψ.

Exemplo 1 (Exemplo de definição de regra de ponte)**.**

$$\frac{DC : \psi \ and \ BC : \ not \ \psi}{IC : \psi}$$

Contextualizando a regra ao cenário de exemplo desenvolvido nesta proposta e considerando que o contexto de crenças seja definido como: *posicao(0,0)* e o contexto de desejos como: *posicao(10,10)* ao executar a regra do Exemplo 1 o contexto de intenções passa a ser: *posicao(10,10)*. Mostrado o funcionamento de uma regra de ponte e seguindo a especificação de (GELAIM et al., 2018), a Definição 6 mostra o conjunto de regras de ponte BDI que serão implementadas neste trabalho.

Definição 6 (Definição das regras de ponte para o comportamento BDI)**.** *Seja* Δ_{br} *o conjunto de regras de ponte. Considerando um agente*

BDI, Δ_{br} *é definido como:*

$$\Delta_{br} = \{$$

$$\frac{CC:\ sense(\varphi)\ and\ PC:\ plan(\varphi, \alpha, Pre, Post, c_a)\ and\ DC:\ \varphi}{BC:\ \varphi} \tag{4.7}$$

$$\frac{DC:\ \varphi\ and\ BC:\ not\ \varphi}{IC:\ \varphi} \tag{4.8}$$

$$\frac{PC:\ plan(\varphi, \alpha, Pre, Post, c_a)\ and\ IC:\ \varphi\ and\ BC:\ Pre}{CC:\ \alpha} \tag{4.9}$$

$$\}$$

A regra 4.7 é responsável por copiar as percepções sense(φ) do contexto de comunicação (CC) para o contexto de crenças (BC), considerando que φ esteja relacionado as pré condições de algum plano que satisfaça algum desejo do agente. A regra 4.8 é responsável por sintetizar uma intenção (IC) φ, e segue a mesma especificação já mostrada no Exemplo 1. A regra 4.9 é responsável por elencar um plano para satisfazer uma intenção φ passando para o contexto de comunicação uma ação α que vai ser executada por um atuador.

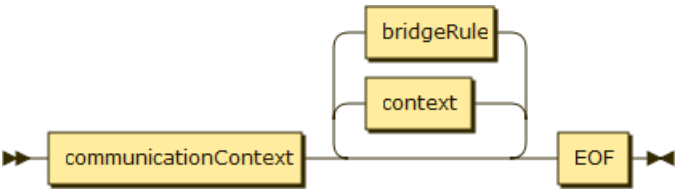
4.3 LINGUAGEM

A linguagem é descrita por uma gramática definida em (GELAIM et al., 2018). Por meio da linguagem é possível realizar o *parser* dos arquivos fontes que definem um agente. Nessa seção são mostradas as principais produções da gramática que está disponível na íntegra no Anexo A.

O agente proposto é composto por contextos, sendo obrigatório o contexto de comunicação, e regras de ponte. Para isso é definida a regra de produção *agent* mostrada na Figura 6.

Um contexto é definido como funcional ou de estado mental, nomeado na gramática como *logicalContext*, conforme mostra a Figura 7. Contextos de estado mental, representados na Figura 8, são compostos por um nome que o identifica e suas fórmulas. O nome pode ser primitivo (*'beliefs'*, *'desires'* ou *'intentions'*) ou customizado para contextos criados pelo desenvolvedor do agente e deve iniciar com o símbolo‘_’.

Figura 6 – Produção *agent*



As fórmulas podem ser proposicionais ou de primeira ordem seguindo as lógicas do agente.

Figura 7 – Produção *context*

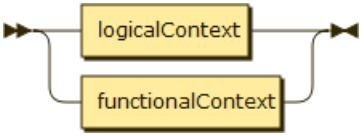
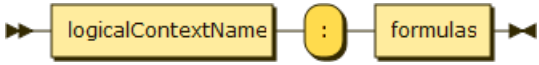


Figura 8 – Produção *logicalContext*



O Exemplo 2 ilustra o uso de contextos mentais, considerando o cenário proposto neste trabalho. Onde o agente acredita estar na posição (0,0), deseja estar na posição (10,10) e desviar dos obstáculos e tem a intenção de estar na posição (10,10).

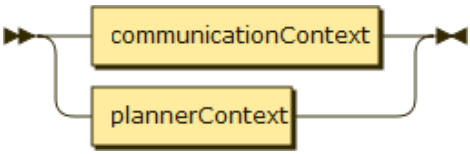
Exemplo 2. Uso de contextos lógicos por meio da gramática.

```
beliefs:
 position(0,0).
desires:
 position(10,10).
 not obstacle.
intentions:
 position(10,10).
```

Contextos funcionais, podem ser de comunicação ou planejamento conforme mostra a Figura 9.

Um contexto de comunicação é derivado por sensores e atuadores, que são definidos da mesma forma, por um identificador e uma

Figura 9 – Produção *functionalContext*



implementação. A representação do contexto de comunicação utilizado no cenário, com dois sensores e dois atuadores, é mostrado no Exemplo 3. No exemplo, são definidos dois sensores e dois atuadores, por meio dos predicados reservados *sensor* e *actuator*. Usando como base o primeiro sensor, o argumento *"positionSensor"* representa a identificação e o argumento *"sensor.PositionSensor"* representa a implementação do sensor. Os demais sensores e atuadores seguem este mesmo padrão.

Exemplo 3. Representação do contexto de comunicação.

```
communication:
    sensor("positionSensor", "sensor.PositionSensor").
    actuator("next", "actuator.NextSlot").
    actuator("toDodge", "actuator.BurnGarbage").
```

A produção *plannerContext* deriva planos e ações. Um plano possui um objetivo, um conjunto de ações, pré condições, pós condições e um custo que é de uso opcional. O Exemplo 4 mostra o uso de dois planos por meio da linguagem.

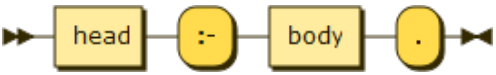
Exemplo 4. Representação do contexto de planejamento.

```
planner:
        plan(check(slots), [action(next(slot))],
            [not obstacle, not end], _).
        plan(check(slots), [action(toDodge(garbage))],
            [obstacle, not end], not obstacle).
```

Outro elemento importante para a definição do agente são as regras de ponte. Considerando que a arquitetura permite ao desenvolvedor adicionar novas regras em seu agente, a grámatica deve possui uma produção que permita essa operação. Uma regra de ponte é composta por uma cabeça (*head*) e um corpo (*body*), conforme mostra a Figura 10.

A produção *head* representa o contexto em que se desejar adicionar uma teoria. O corpo da regra de ponte, representado pela produção

Figura 10 – Produção *bridgeRule*



body pode conter um ou mais contextos, que são ligados pelos conectores *and* ou *or*. Considerando um agente BDI e as regras formalizadas na Definição 6, o Exemplo 5 mostra a definição das regras utilizando a linguagem.

Exemplo 5. Representação das regras de ponte por meio da linguagem.

```
1  !beliefs X :- communication sense(X)
2          & planner plan(Y,_,Z,_)
3          & planner member(X, Z) | planner member(not X, Z)
4          & desires X.
5
6  !intentions X :- desires X
7              & beliefs not X & intentions not X.
8
9  !communication Actions :- planner plan(0, Actions, Pre, Post, _)
10             &  intentions 0 & beliefs Pre.
```

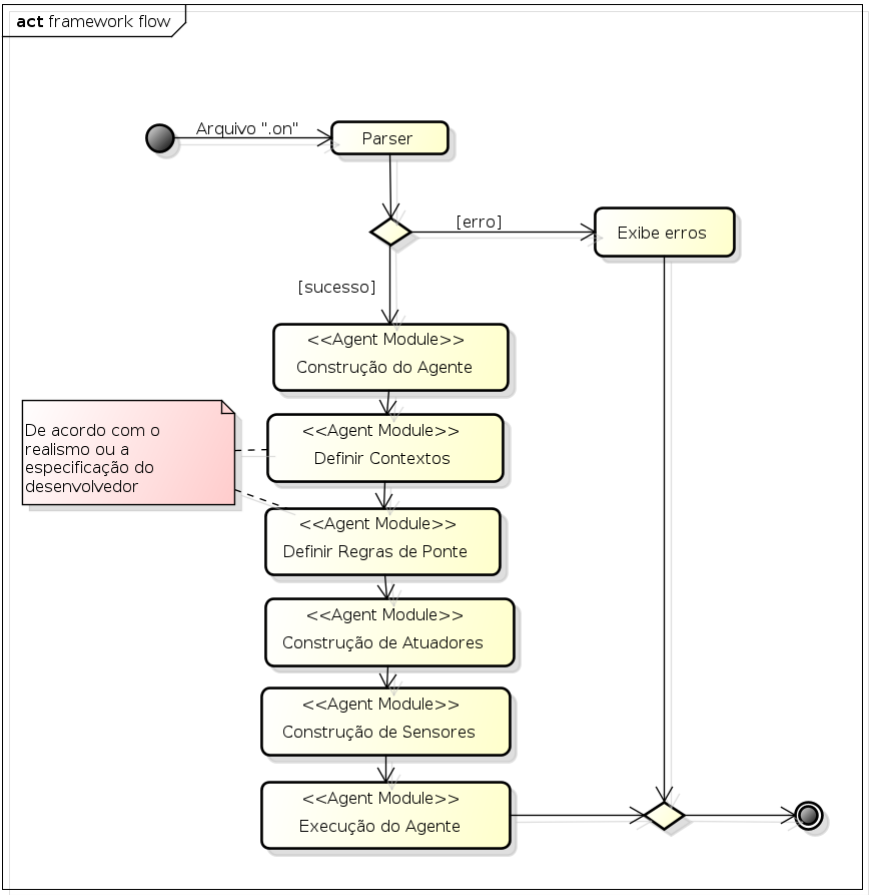
Com base na definição da arquitetura e da linguagem a seção 4.4 mostra a construção do *Framework* proposto neste trabalho, sendo a contribuição central deste trabalho.

4.4 *FRAMEWORK*

Com o objetivo de permitir a operacionalização da linguagem e a execução do agente, nessa seção é apresentada uma proposta de arcabouço de *software* denominada como *framework*. Sua implementação, realizada em JAVA, é a principal contribuição deste trabalho.

Para simplificar o desenvolvimento e desacoplar os componentes, a estrutura a ser implementada neste trabalho segue a especificação de (GELAIM et al., 2018), e é dividida em dois componentes principais: *Parser* e *Agent*. A Figura 11 mostra a relação entre os componentes e o fluxo geral do *framework*. A especificação de cada componente implementado será feita nas seções seguintes.

Figura 11 – Fluxo geral de execução do framework



Fonte: Gelaim et al. (2018).

4.4.1 Parser

O componente *Parser* é responsável pela transformação dos arquivos fontes do agente em uma estrutura de dados executável. Para isso ele utiliza a gramática da linguagem apresentada na seção 4.3. A

geração do Parser é feita pela ferramenta Antlr [1] que fornece diversas classes para a representação e tratamento de texto e gramáticas.

A entrada é feita por um único arquivo em formato *.on* contendo a definição do agente que foi mostrada na seção 4.3. O processo de tradução é dividido em duas fases: a análise léxica e a análise sintática, sendo essa a primeira etapa da execução do agente. Caso haja erros léxicos ou sintáticos o desenvolvedor é notificado pela classe *VerboseListener*, mostrada no Código 1.

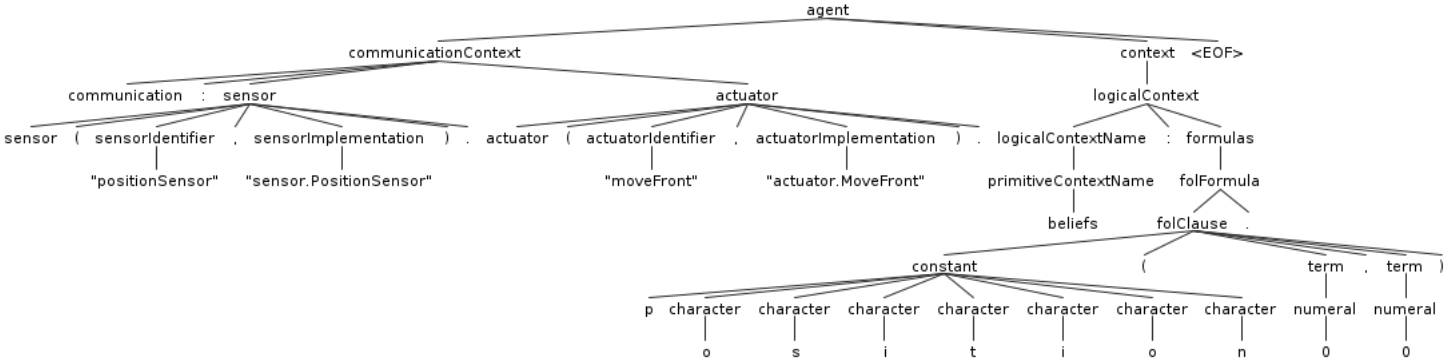
Código 1. Representação da classe *VerboseListener*

```
1  public class VerboseListener extends BaseErrorListener {
2
3          @Override
4          public void syntaxError(Recognizer<?, ?> recognizer,
5          Object offendingSymbol, int line,
6                   int charPositionInLine,
7                          String msg, RecognitionException e) {
8                  List<String> stack = ((Parser) recognizer)
9                  .getRuleInvocationStack();
10                 Collections.reverse(stack);
11                 System.err.println("rule stack: " + stack);
12                 System.err.println("line " + line + ":"
13                 + charPositionInLine + " at " + offendingSymbol
14                 + ": " + msg);
15         }
16 }
```

Em caso de sucesso é gerada uma árvore sintática que será utilizada para a execução do agente. Contextualizando o cenário proposto, a Figura 12 mostra parte da árvore gerada utilizando o exemplo da seção 4.3.

[1] http://www.antlr.org/

Figura 12 – Árvore Sintática



4.4.2 Agente

Similar a definição formal de um agente como sistema multi-contexto, a representação do agente como um *software* é feita por decomposição modular usando contextos e regras de pontes. Um contexto, é uma unidade funcional de *software* e uma regra de ponte é utilizada para a troca de informação entre as unidades. Para representar um Agente, é definido no *framework* a classe *Agent* que contém referencias para instâncias de contextos e regras de ponte. Após a execução do *parser* é criada uma instância de Agente e sua execução é inicializada.

4.4.2.1 Contextos

Um contexto é generalizado no *framework* por meio da *inteface ContextService*, mostrada na Figura 13, que estabelece métodos para: adição, atualização e busca de teoria, que neste trabalho é representada por uma base de conhecimento; verificação de uma teoria e nome do contexto. Sob a perspectiva da engenharia de *software* um contexto representa um componente que encapsula regras internas e por meio de seus métodos regras de ponte integram conhecimento entre diferentes unidades. A teoria, representada pela classe *Theory*, é implementada na biblioteca tuProlog [2] que fornece um ambiente prolog , sobre o qual é executado o raciocínio utilizando a (*Java-Virtual-Machine*) .

Considerando um cenário BDI e o contexto de crenças representado no Exemplo 6, ao executar o método *verify("obstacle")* da implementação do contexto de crenças, é retornado o valor 'false' considerando que segundo a teoria do contexto, não existe um obstáculo no estado atual.

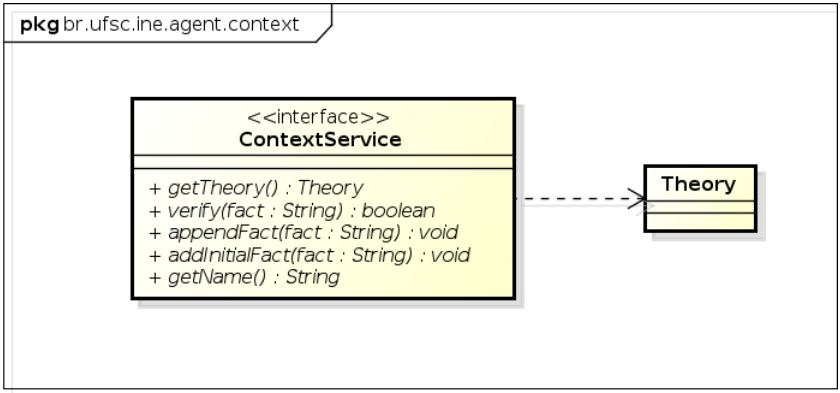
Exemplo 6. Exemplo de contexto de desejos.

```
1 beliefs:
2  position(10,10).
3  not obstacle.
```

Este trabalho implementa por padrão os contextos BDI (crenças, desejos e intenções) e o contexto de comunicação e planejamento. Apesar da linguagem permitir a criação de novos contextos pelo desenvolvedor do agente não será aplicada semântica a estes contextos.

[2] http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome

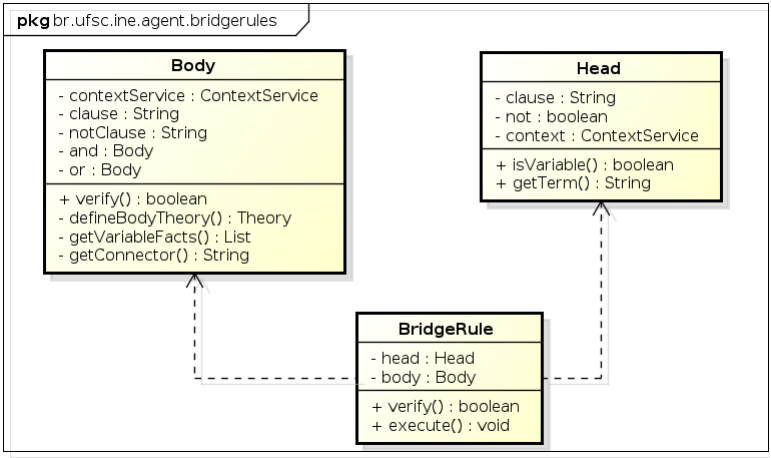
Figura 13 – Interface ContextService



4.4.2.2 Regras de ponte

Com o objetivo de integrar conhecimento entre os contextos, foi criada uma estrutura de dados para representar as regras de ponte, com base na especificação feita em (GELAIM et al., 2018). A Figura 14 mostra o Diagrama de classes, com os principais métodos, criado para a definição de uma regra de ponte no *framework*.

Figura 14 – Diagrama de classe representando uma regra de ponte



Com base na estrutura de regras de ponte é executado o Algoritmo 2, especificado em (GELAIM et al., 2018), que representa o raciocínio sobre as regras de ponte.

Algoritmo 2: Raciocínio sobre as regras de ponte

```
1  bt ← defineTeoriaDoCorpo();
2  test ← defineQuestao();
3  s ← prolog.verificar(test, bt);
4  if s.sucesso then
5  |  hc ← buscaClausulaDaCabeca();
6  |  t ← s.buscaTermos(hc);
7  |  if t.possuiSolucao then
8  |  |  adicionaSolucaoNaCabeca(t.solucoes);
9  |  end
10 end
```

Para auxiliar o entendimento das regras de ponte e sua execução, o Código 2 mostra regra de ponte 4.8 da seção 4.2.3 implementada no *framework*.

Código 2. Exemplo de regra de ponte implementada no *framework*.

```
Head head = Head.builder()
    .context(intentionsContext)
    .clause("X").build();

Body body = Body.builder()
                .context(desiresContext).clause("X")
                        .and(Body.builder()
                                .context(beliefsContext)
                                .notClause("X").build())
                        .build();

BridgeRule r2 =  BridgeRule.builder()
                .head(head)
                .body(body)
                .build();

r2.execute();
```

Por padrão o *framework* implementa as regras detalhadas na seção 4.2.3. Para as regras adicionadas pelo desenvolvedor não será

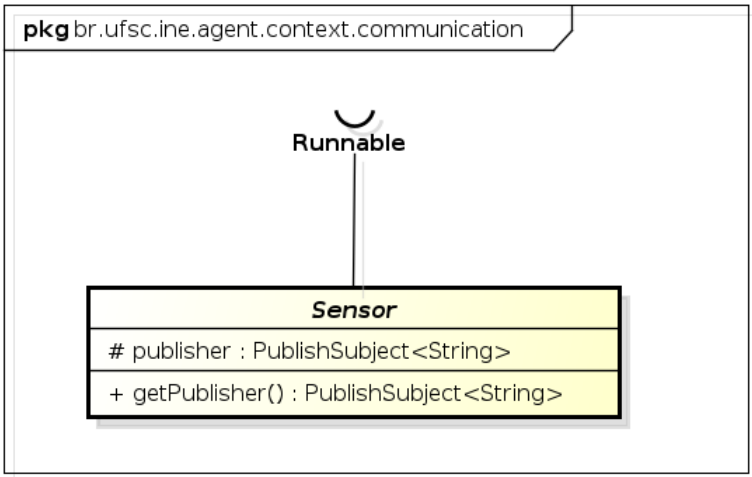
aplicado semântica durante a execução.

4.4.3 Comunicação com o ambiente

A comunicação do agente com seu ambiente ou com outros agentes é feita pelo contexto de comunicação por meio de sensores e atuadores, não acoplados a uma estrutura de ambiente. Esta característica, permite a operação em ambientes de diferentes topologias. Ou seja, fazendo uso da estrutura dos sensores e atuadores é possível integrar o agente com diferentes tipos de ambientes implementados em diferentes linguagens.

Um sensor deve retornar um valor literal conforme as lógicas do Agente mostradas na seção 4.2.1. Sua implementação é feita pelo desenvolvedor seguindo a especificação do *framework* que estabelece a classe abstrata *Sensor*, conforme mostra a Figura 15. Sob o ponto de vista da arquitetura, um sensor contém um publicador de literais, seguindo o paradigma da programação reativa[3].Considerando que o agente pode ter vários sensores que podem capturar percepções de forma paralela um Sensor é tratado como uma *Thread* implementando a interface *Runnable*.

Figura 15 – Classe *Sensor*



[3]http://reactivex.io/documentation/subject.html

Internamente, o publicador será assinado pelo Contexto de comunicação. Ao receber uma percepção ela é persistida no contexto de comunicação e o conjunto de regras de ponte é executado, sintetizando um ciclo de raciocínio.

A implementação de um sensor de posição, considerando o agente desenvolvido nesta proposta, é mostrada no Código 3. O método *run* é invocado pelo agente na inicialização disparando uma *Thread*.

Código 3. Implementação de um sensor de posição

```
1  import br.ufsc.ine.agent.context.communication.Sensor;
2  import rx.subjects.PublishSubject;
3
4  public class PositionSensor extends Sensor {
5
6   public static final PublishSubject<String> position
7      = PublishSubject.create();
8
9   @Override
10  public void run() {
11     position.subscribe(super.getPublisher());
12  }
13 }
```

Para alterar o ambiente ou enviar mensagens para outras entidades, o agente utiliza seus atuadores. Um atuador é definido pelo *framework* como uma classe abstrata, mostrada na Figura 16. Considerando que cada ciclo é executada uma única ação diferente de um sensor um atuador não é uma *Thread*.

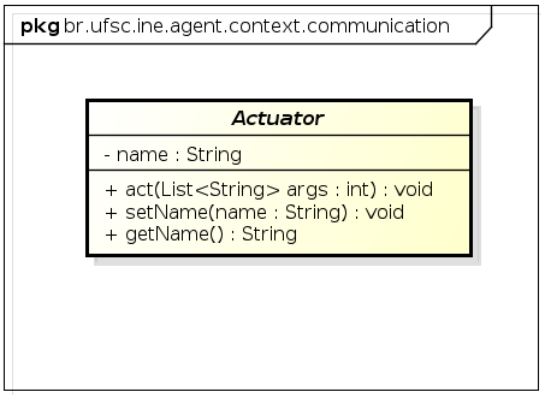
Para executar uma ação o contexto de planejamento invoca o método *act* passando de forma opcional uma lista de argumentos. Seu uso é ilustrado no Código 4.

Código 4. Implementação de um atuador

```
1  public class NextSlot  extends Actuator {
2      public void act(List<String> list) {
3          Platform.runLater(new Runnable() {
4              @Override
5              public void run() {
6                  Main.nextSlot();
7              }
8          });
```

```
9         }
10    }
```

Figura 16 – Classe *Actuator*



4.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou o modelo de agente como sistema SMC definido por (GELAIM et al., 2018), e com base neste modelo as principais classes e interfaces implementadas no *framework*. Ressalta-se que o exemplo utilizado visa apenas conduzir o leitor, dando uma ideia da sintaxe da linguagem e da arquitetura do framework. O código completo do agente, implementado na linguagem proposta, será mostrado no capítulo 5.

5 RESULTADOS E AVALIAÇÃO

Neste capítulo são apresentados os principais resultados obtidos com a implementação do *framework*. O enfoque da avaliação, alinhado com os objetivos do trabalho, está na execução de um único agente implementado na linguagem e mostrado na seção 5.1. Paralelamente, na seção 5.2 é realizado um experimento com o uso de contextos e regras de ponte para enfatizar e demonstrar o potencial de expressividade do *framework*.

5.1 AVALIAÇÃO DO CASO DE USO

Conforme o cenário proposto no capítulo 4, seção 4.2, suponha que um Agente é responsável em deslocar em um ambiente que pode conter obstáculos. Caso encontre um obstáculo, o agente deve desviar. O Código 5 mostra a definição deste agente por utilizando a linguagem implementada no *framework*.

Código 5. Representação do agente por meio da linguagem SMC

```
1  communication:
2      sensor("positionSensor", "sensor.PositionSensor").
3      actuator("next", "actuator.NextSlot").
4      actuator("toDodge", "actuator.ToDodge").
5
6  desires:
7          check(slots).
8          not obstacle.
9
10 intentions:
11         end.
12
13 planner:
14         plan(check(slots),
15             [action(next(slot))], [not obstacle, not end], _).
16
17         plan(check(slots),
18             [action(toDodge(garbage))],
19             [obstacle, not end], not obstacle).
```

Conforme pode ser observado, na linha 1 é criado o contexto de

comunicação composto por um sensor (*'positionSensor'*) e dois atuadores (*'next'*, *'toDodge'*). Na linha 6, é criado o contexto de desejos contendo dois termos *'check(slots).'* e *'not obstacle.'* Na linha 10, é criado o contexto de intenções com uma única intenção *'end.'*. Por fim, na linha 13 é criado o contexto de planejamento contendo dois planos definidos nas linhas 14 e 17.

O atuador *'toDodge'* é mostrado no Código 6. Ele especializa a classe *Actuator* que define um atuador no *framework*. Na linha 8, é feita a chamada do método da classe *Main* que, neste exemplo, representa o ambiente. Considerando que o atuador é desacoplado do ambiente, diferente da estratégia utilizada na linguagem Jason, pode-se dizer que a integração com diferentes ambientes é facilitada. Isso também vale para os sensores, que seguem a mesma estratégia de implementação.

Código 6. Implementação de um atuador

```
1  public class ToDodge  extends Actuator {
2
3      public void act(List<String> args) {
4
5          Platform.runLater(new Runnable() {
6              @Override
7              public void run() {
8                  Main.toDodge();
9              }
10         });
11
12     }
13 }
```

O sensor *'positionSensor'* e o atuador *'next'* declarados na definição do agente foram anteriormente detalhados no capítulo 4. Considerando que a implementação do ambiente foge do escopo deste trabalho, a representação da classe *Main* não será detalhada. Por outro lado, é importante destacar que é perfeitamente possível integrar os sensores e atuadores com outras linguagens. Por exemplo, utilizando TCP-IP é possível receber e enviar sinais para um ambiente de realidade virtual implementado em Unity [1] ou com ambientes reais enviando e recebendo sinais de câmeras de trânsito.

Com o objetivo de quantificar o tempo de execução o uso de memória do agente foram feitas dez execuções do Código 7. Na linha 1

[1] https://unity3d.com/pt

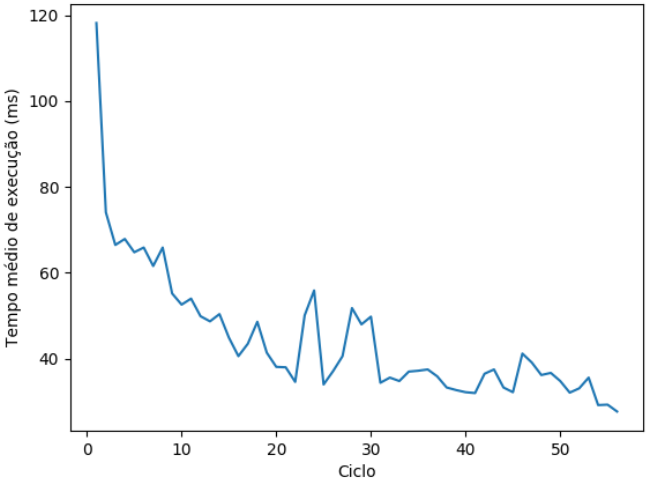
é instanciada uma árvore sintática, passando como parâmetro o arquivo de definição do agente *"agent.on"*. As linhas 2 e 3 definem a criação de um *walker* para percorrer os nodos da árvore e possibilitar a execução do agente. Na linha 5 é criada uma instancia do agente. Para facilitar a coleta do tempo de execução, de cada ciclo de raciocínio, foi adicionado na linha 6 um arquivo de *profiling*. Por fim, na linha 7 é feita a execução do agente.

Código 7. Código de execução do agente

```
1       ParseTree tree = getParseTree("agent.on");
2       ParseTreeWalker walker = new ParseTreeWalker();
3       walker.walk(new AgentWalker(), tree);
4
5       Agent agent = new Agent();
6       agent.setProfilingFile("esultados.csv");
7       agent.run(agentWalker);
```

Cada uma das dez execuções teve 56 ciclos de raciocínio, o tempo de médio de execução por ciclo é mostrado na Figura 17.

Figura 17 – Tempo médio de execução por ciclo

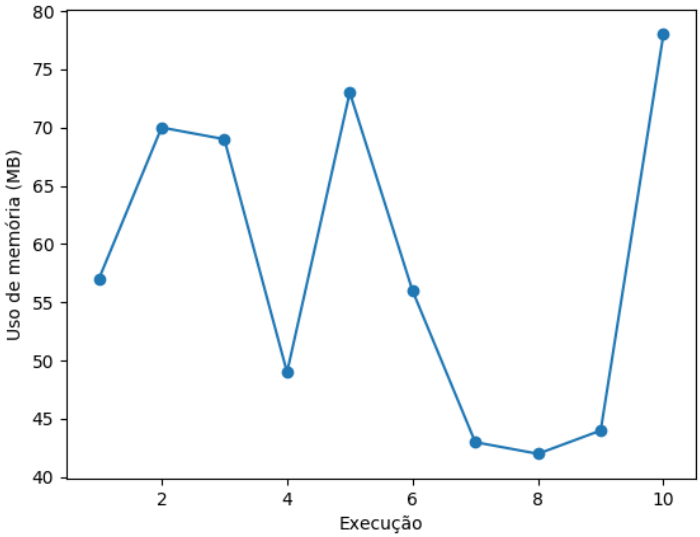


O maior tempo médio de execução foi do primeiro ciclo com 118.2 ms, o menor tempo médio foi do ciclo 56 com 27.7 ms. O tempo médio

geral de todos os ciclos foi de 44.46 ms com um desvio padrão de 15.26 ms. Conforme pode ser observado existe um custo inicial alto na execução devido a criação de instancias de objetos JAVA para representar os contextos e regras de ponte. Essa instanciação é realizada uma única vez no primeiro ciclo de raciocínio conforme o agente necessita utilizar os contextos. Além disso a tendência média é o tempo de execução do ciclo diminuir, isso se dá devido as otimizações internas feitas pela JVM após a invocação repetida dos mesmos métodos.

Um outro fator importante na avaliação do agente é o uso de memória. Para coletar o uso de memória da execução, foi utilizada a ferramenta *JVisualVM* incluída no pacote da JVM. Considerando as 10 execuções do cenário que foram realizadas, a Figura 18 mostra o uso máximo de memória por execução, cada execução é composta igualmente por 56 ciclos de raciocínio. O valor máximo de consumo de memória é de 78 MB, o valor mínimo é de 42 MB e o valor médio é de 58.1 MB.

Figura 18 – Uso de memória por execução



Considerando que a execução do ambiente foi feita no mesmo processo da JVM, este uso de memória representa a execução do agente e do ambiente. Devido a este fator a a curva de utilização de memó-

ria não se manteve estável. Por outro lado a quantidade de memória utilizada para a execução do agente e do ambiente é satisfatória. Os testes foram executados em uma máquina com sistema operacional Linux Mint 18 Sarah com 8 GB de memória. A versão da JVM utilizada foi a *openjdk version 1.8.0_162*. O processador utilizado foi o Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz.

Com base nos resultados mostrados, pode-se dizer que a abordagem implementada, representando um agente como Sistema Multi-Contexto, é aceitável considerando os objetivos do trabalho. Por outro lado, existem pontos a serem melhorados, por exemplo o tempo de custo inicial do agente que está atrelado a criação dos contextos e das regras de ponte. Além disso é necessário realizar testes de estresse com os sensores e atuadores para avaliar o uso do agente em ambientes de muitas percepções. As sugestões de melhorias serão melhor detalhadas no capítulo 6, na seção de trabalhos futuros. Para maiores detalhes referente a implementação, o exemplo completo está disponível no repositório do Github [2].

5.2 AVALIAÇÃO DE CONTEXTOS E REGRAS DE PONTE

Esta seção tem por objetivo avaliar o uso de contextos e de regras de ponte por meio do *framework* implementado. A criação de novos contextos e regras pontes permitem flexibilizar o modelo, adicionando um maior potencial de expressividade. De forma mais específica, usando uma regra de ponte é possível raciocinar sobre diferentes contextos que podem representar qualquer estado mental, por exemplo um contexto emocional.

Para permitir a criação de um contexto qualquer, fora da arquitetura BDI, foi criada a classe *CustomContext* que implementa a interface *ContextService* especificada no capítulo 4. Para avaliar seu uso, foram criados 3 contextos: *_X*, *_Z*, *_Y*, conforme mostra o Código 8.

Código 8. Criação de novos contextos por meio do *framework*.

```
1 CustomContext _x = new CustomContext("X");
2 CustomContext _y = new CustomContext("Y");
3 CustomContext _z = new CustomContext("Z");
4
5 _y.appendFact("a.");
6 _y.appendFact("b.");
```

[2] https://github.com/valdirluiz/sigon-examples/tree/exemplo-cenario

```
7  _y.appendFact("c.");
8
9  _z.appendFact("b.");
10 _z.appendFact("c.");
11 _z.appendFact("d.");
```

O objetivo do experimento é copiar a intersecção dos elementos dos contextos $\_Y$ e $\_Z$ para o contexto $\_X$ que possui um estado inicial vazio. Para isso, foi definida a regra de ponte mostrada no Código 9.

Código 9. Regra de ponte do experimento.

```
1 Head xHead = Head.builder().context(_x).clause("T").build();
2 Body yBody = Body.builder().context(_y).clause("T").build();
3 Body zBody = Body.builder().context(_z).clause("T").build();
4
5 // !_X T :- _Y T & _Z T
6 BridgeRule bridgeRule = BridgeRule.builder().head(xHead)
7     .body(yBody.and(zBody))
8     .build();
```

O *log* da execução do experimento é o seguinte:

```
Estado do contexto Y antes execução da regra:
a.
b.
c.

Estado do contexto Z antes execução da regra:
b.
c.
d.

Estado do contexto X após execução da regra de ponte:
b.
c.
```

Conforme pode se observado ao final de execução, após a execução da regra de ponte, o contexto X possui a interseção dos elementos dos contextos Y e Z.

Ao alterar o operador da regra de ponte para *or*, conforme mostra o código 10, o resultado da operação passa a ser a união dos elementos dos contextos $\_y$ e $\_z$.

Código 10. Regra de ponte do experimento.

```
Head xHead = Head.builder().context(_x).clause("T").build();
Body yBody = Body.builder().context(_y).clause("T").build();
Body zBody = Body.builder().context(_z).clause("T").build();

// !_X T :- _Y T | _Z T
BridgeRule bridgeRule = BridgeRule.builder().head(xHead)
    .body(yBody.or(zBody))
    .build();
```

O resultado da execução do código 10 é o seguinte:

```
Estado do contexto Y antes execução da regra:
a.
b.
c.

Estado do contexto Z antes execução da regra:
b.
c.
d.

Estado do contexto X após execução da regra de ponte:
a.
b.
c.
d.
```

Apesar do cenário implementado ser simples, por meio desta arquitetura é possível modelar e executar diferentes formas de raciocínio. O código completo do experimento está disponível no repositório [3] do github.

Por fim, pode-se dizer que os resultados alcançados estão dentro dos objetivos estabelecidos. Por meio do *framework* desenvolvido é possível implementar um agente BDI, fazendo uso de contextos, regras de ponte, sensores e atuadores. Paralelamente, o *framework* também permite a execução de diferentes tipos de raciocínio sem um grande custo de implementação, conforme foi mostrado pelos experimentos com regras de ponte e contextos.

[3] https://github.com/valdirluiz/sigon-examples/blob/tcc-valdir/src/avaliacao/contextos/AvaliacaoContextos.java

6 CONCLUSÃO

Neste trabalho foi apresentada a implementação de um *framework* para a execução de agentes BDI vistos como um Sistema Multi-Contexto. O *framework* implementado seguiu a abordagem de sistemas Multi-Contexto, utilizando como base o modelo proposto por Gelaim et al. (2018). Foram criadas estruturas de *software* para representar um agente por meio de contextos e regras de ponte. Para permitir a execução do agente, com o auxílio da ferramenta ANTLR (PARR, 2013) foi implementado um *parser* descendente recursivo utilizando como base a gramática definida em (GELAIM et al., 2018). No contexto de comunicação, foi criada uma estrutura para a representação e implementação de sensores e atuadores que permitem a integração do agente com diferentes tipos de ambiente ou mesmo com outros agentes.

Por ser baseado em um modelo de agente como SMC, o *framework* permite e favorece a representação de diferentes arquiteturas de agentes. Com o objetivo de demonstrar seu uso, foi proposto um cenário completo de um agente construído sobre uma arquitetura BDI. Paralelamente a isso, foi apresentado um exemplo de uso de regras de ponte e contextos, com o objetivo de mostrar o potencial de expressividade do *framework*.

A literatura apresenta diversos modelos de agente como SMC. Além disso são definidos diferentes arquiteturas de *software* para o desenvolvimento e a operacionalização de agentes, por exemplo JASON e 3APL, ambos mostrados no capítulo 3. Por outro lado, existe uma deficiência na operacionalização de agentes como SMC, as abordagens apresentadas são restritamente teóricas o que dificulta a realização de testes ou a criação de protótipos. Frente a isso, este trabalho contribui na implementação de um *framework* para a execução e prototipação de agentes baseados em SMC. Com o *framework* desenvolvido se torna possível a operacionalização de agentes modelados baseados em um SMC. A partir desta modelagem, a definição e criação de novas arquiteturas de agente, utilizando contextos e regras de ponte, também é facilitada, aumentando o grau de expressividade do agente. O código fonte está disponível em sua integridade no repositório público do GitHub [1].

[1] https://github.com/sigon-lang/sigon-lang

6.1 TRABALHOS FUTUROS

Por se tratar de uma primeira versão de implementação do *framework*, esta pesquisa proporciona várias sugestões de trabalhos futuros. O framework implementado permite a criação e execução de um único agente por vez. A abordagem de sistemas multi agente pode ser melhor explorada em trabalhos futuros.

Apesar da linguagem permitir a criação de novos contextos e novas regras de ponte, fora do escopo BDI e dos contextos de comunicação e planejamento, não é executada nenhuma semântica sobre estes contextos. Uma abordagem futura pode permitir a integração destes novos contextos e regras de ponte com o modelo BDI já implementado por padrão no *framework*.

Além disso, existem aspectos a serem melhorados na implementação do *parser*. Na versão atual não é feita nenhuma validação semântica sobre o código fonte do agente, essas validações podem ser tratadas e analisadas em trabalhos futuros. Também é necessário executar testes de stress envolvendo os atuadores e sensores para verificar o comportamento do agente em ambientes muito dinâmicos.

Por fim, esta pesquisa focou no desenvolvimento de um *framework* para a execução de agentes usando representações de conhecimento baseadas em lógica de primeira ordem e proposicional. A integração e criação de contextos com diferentes tipos de raciocínio, por exemplo com ontologias, pode ser implementada em trabalhos futuros.

REFERÊNCIAS

BORDINI, R. H.; HÜBNER, J. F. Bdi agent programming in agentspeak using jason. In: TONI, F.; TORRONI, P. (Ed.). *Computational Logic in Multi-Agent Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 143–164. ISBN 978-3-540-33997-7.

BRATMAN, M. *Intention, plans, and practical reason*. [S.l.]: Harvard University Press, 1987. ISBN 9780674458185.

CASALI, A. *On intentional and social agents with graded attitudes*. Tese (Doutorado) — Universitat de Girona, 2008.

CASALI, A.; GODO, L.; SIERRA, C. Graded BDI models for agent architectures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 3487 LNAI, p. 126–143, 2005. ISSN 03029743.

DASTANI, M. et al. A programming language for cognitive agents goal directed 3apl. In: DASTANI, M. M.; DIX, J.; FALLAH-SEGHROUCHNI, A. E. (Ed.). *Programming Multi-Agent Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 111–130. ISBN 978-3-540-25936-7.

FRANKLIN, S.; GRAESSER, A. Is it an agent, or just a program?: A taxonomy for autonomous agents. In: _____. *Intelligent Agents III Agent Theories, Architectures, and Languages: ECAI'96 Workshop (ATAL) Budapest, Hungary, August 12–13, 1996 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. p. 21–35. ISBN 978-3-540-68057-4. <https://doi.org/10.1007/BFb0013570>.

GELAIM, T. Â. *Modelo de agentes e-BDI integrando confiança baseado em sistemas multi-contexto*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2016.

GELAIM, T. ; SILVEIRA, R. A.; MARCHI, J. Towards a model of cognitive agents: Integrating emotion on trust. In: *2015 Fourteenth Mexican International Conference on Artificial Intelligence (MICAI)*. [S.l.: s.n.], 2015. p. 80–86.

GELAIM, T. Ângelo et al. Sigon: A multi-context system framework for intelligent agents. *Expert*

Systems with Applications, 2018. ISSN 0957-4174. <http://www.sciencedirect.com/science/article/pii/S0957417418307000>.

GHIDINI, C.; GIUNCHIGLIA, F. Local models semantics, or contextual reasoning=locality+compatibility. *Artificial Intelligence*, v. 127, n. 2, p. 221 – 259, 2001. ISSN 0004-3702. <http://www.sciencedirect.com/science/article/pii/S0004370201000649>.

GIUNCHIGLIA, F.; SERAFINI, L. Multilanguage hierarchical logics, or: How we can do without modal logics. *Artificial Intelligence*, v. 65, n. 1, p. 29 – 70, 1994. ISSN 0004-3702. <http://www.sciencedirect.com/science/article/pii/000437029490037X>.

HERZIG, A. et al. Bdi logics for bdi architectures: Old problems, new perspectives. *KI - Künstliche Intelligenz*, v. 31, n. 1, p. 73–83, Mar 2017. ISSN 1610-1987. <https://doi.org/10.1007/s13218-016-0457-5>.

PARR, T. *The definitive ANTLR 4 reference*. [S.l.]: Pragmatic Bookshelf, 2013.

RAO, A. S.; GEORGEFF, M. P. *Modeling Rational Agents within a BDI-Architecture*. 1991.

RAO, A. S.; GEORGEFF, M. P. Bdi agents: From theory to practice. In: *ICMAS*. [S.l.: s.n.], 1995. p. 312–319.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 3rd. ed. [S.l.: s.n.], 2009.

SABATER, J. et al. Using multi-context systems to engineer executable agents. In: JENNINGS, N. R.; LESPÉRANCE, Y. (Ed.). *Intelligent Agents VI. Agent Theories, Architectures, and Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 260–276. ISBN 978-3-540-46467-9.

SILVEIRA, R. et al. Towards a model of open and reliable cognitive multiagent systems: Dealing with trust and emotions. *ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal*, v. 4, n. 3, 2016. ISSN 2255-2863.

WOOLDRIDGE, M. *Reasoning About Rational Agents*. [S.l.]: MIT Press, 2000.

WOOLDRIDGE, M. Intelligent agents: The key concepts. In: _____. *Multi-Agent Systems and Applications II: 9th ECCAI-ACAI /*

EASSS 2001, AEMAS 2001, HoloMAS 2001 Selected Revised Papers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 3–43. ISBN 978-3-540-45982-8. <https://doi.org/10.1007/3-540-45982-0$_1$>.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, Cambridge University Press, v. 10, n. 2, p. 115–152, 1995.

APÊNDICE A – Artigo

Implementação de um framework para o desenvolvimento de agentes Sigon como sistema multi-contexto

Valdir Luiz Hofer Arnhold[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88.040-900 – Florianópolis – SC – Brazil

valdir.luiz@grad.ufsc.br

Abstract. *Artificial intelligence (AI) is subject of study in several areas of knowledge. Throughout history, it has gone through different stages and contributed for innovating on different fronts. Today, the AI is used as support for the development of different tools directly affecting the user experience and his way to deal with tecnology. One of the fronts of AI is dedicated to the study and development of models and architectures for the development of cognitive agents. Agent-based systems can be used in different contexts, a fact that emphasizes the importance of the development of new methods in order to enable their creation or prototyping. The literature presents several approaches and methods for the development of agent models including Multi-Context Systems. This work presents the development of a framework to conceive BDI agents as a Multi-Context System. As main results, it is demonstrated the execution of a BDI agent seen as a Multi-Context system and the use of bridge rules to reason about different contexts.*

Resumo. *A inteligência artificial (IA) é tema de estudo em diversas áreas de conhecimento. Ao longo da história, ela passou por diferentes etapas e contribui inovando em diferentes frentes. Hoje, a IA é utilizada como suporte para o desenvolvimento de diferentes ferramentas afetando diretamente a vivência e a experiência de uso dos usuários. Uma das frentes da IA se dedica ao estudo e desenvolvimento de modelos e arquiteturas para a criação de agentes cognitivos. Sistemas baseados em agentes, podem ser utilizados em diferentes contextos, fato que ressalta a importância do desenvolvimento de novos métodos a fim de possibilitar sua criação ou prototipação. A literatura, apresenta diversas abordagens e métodos para a criação de modelos de agente entre elas a de Sistemas Multi-Contexto. Este trabalho, apresenta a criação de um* framework *para permitir o desenvolvimento e criação de agentes BDI vistos como um Sistema Multi-Contexto. Como principais resultados, é demostrada a execução de um agente BDI visto como sistema Multi-Contexto e o uso de regras de ponte para raciocinar sobre diferentes contextos.*

1. Introdução

Um computador, em sua forma original, não possui habilidade para decidir por conta própria o que fazer, cada ação deve estar programada de forma explícita em um programa [Russell and Norvig 2009]. Em muitos casos, aplicações tradicionais, sem inteligência e raciocínio, satisfazem seus usuários conseguindo executar todas as tarefas propostas.

Porém, com o avanço da tecnologia, é exigido cada vez mais que determinadas aplicações possam pensar e decidir por si mesmas [Wooldridge 2002, p.14]. Esta necessidade não é satisfeita com o uso da computação em sua forma tradicional. Frente a isso, é introduzido o conceito de IA (Inteligência Artificial), um campo de estudo interdisciplinar que busca atuar na resolução de problemas complexos e não tradicionais.

A Inteligência Artificial faz uso dos conhecimentos de diferentes áreas, por exemplo da Biologia, da Antropologia, da Psicologia e da Matemática, para a construção de seus modelos. Devido a essa generalidade, sua pesquisa é dividida em diferentes campos de atuação, que vão do geral ao específico sendo relevante para qualquer tarefa intelectual [Russell and Norvig 2009]. Um destes campos de estudo busca definir e criar ferramentas para o desenvolvimento de agentes. Um agente, é um sistema computacional inserido em um ambiente e capaz de executar ações autônomas para satisfazer seus objetivos pré-designados [Wooldridge 2002].

Agentes artificiais, alinhados com a pluralidade da inteligência artificial, são utilizados na resolução de diferentes tarefas em diferentes áreas de atuação. Devido a esta amplitude no uso de agentes, surgiram diversas arquiteturas que buscam modelar um agente conforme a necessidade de seu uso. Uma das arquiteturas para a definição de agentes é o modelo (BDI) *Belief-Desire-Intention* fundamentado na teoria do raciocínio prático. Um agente BDI, utiliza um conjunto de crenças, desejos e intenções para a representação de seu conhecimento, e com base neste conhecimento são realizadas ações internas e no mundo [Rao and Georgeff 1995, p.315].

Por outro lado, a pluralidade do uso de agentes em diferentes contextos necessita de diferentes configurações em uma arquitetura. Tipicamente, as linguagens de programação de agentes são restritas devido a questões relacionadas a eficiência de seu ciclo de raciocínio ou a complexidade das lógicas existentes [Herzig et al. 2017, p.79]. Esta natureza acaba inibindo o uso de sistemas baseados em agentes em determinadas soluções computacionais. Apesar da literatura definir modelos genéricos para a modelagem de agentes, há um distanciamento entre a formalidade das lógicas com sua devida implementação [Herzig et al. 2017, Casali 2008, Sabater et al. 2000].

Frentes a estas questões, é apresentado em [Ângelo Gelaim et al. 2018] um modelo para a definição de agentes vistos como SMC (Sistema Multi-Contexto). O modelo permite a definição de um agente genérico que pode ser facilmente adaptado e especializado conforme a necessidade de seu uso. Com base neste modelo, este artigo apresenta um *framework* com componentes de *software* para permitir a execução e operacionalização de um SMC. O SMC implementado foi utilizado para a representação de um agente baseado em uma arquitetura BDI. Porém, é possível criar diferentes configurações com contextos e regras de ponte.

Este artigo é organizado da seguinte forma: Na seção 2 são mostrados os trabalhos relacionados que fundamentam a proposta, na seção 3 é mostrada a proposta de desenvolvimento, na seção 4 são analisados os principais resultados, por fim na seção 5 apresenta a conclusão e as sugestões de trabalhos futuros.

2. Trabalhos correlatos

Gelaim, 2016 [Gelaim 2016] apresenta um modelo de agente inspirado em BDI que integra um modelo computacional de emoções. A abordagem utilizada para o modelo é

baseada em um Sistema Multi-Contexto. No modelo proposto, as crenças são graduadas com graus de certeza do agente sobre elas e podem ser rotuladas como especiais. As crenças especiais são chamadas de julgamentos e são utilizadas para intermediar a relação entre confiança e emoções. O modelo é formalmente definido como:

Definição 1 (Modelo de agente (*Gelaim, 2016*))

$$AG = \langle \{BC, DC, IC, PC, TC, EC, CC\}, \Delta_{rp} \rangle \quad (1)$$

onde BC, DC, IC, PC, TC, EC, CC são, respectivamente, os contextos de crenças, desejos, intenções, planejamento, confiança, emoções e comunicação; e Δ_{rp} são as regras de ponte utilizadas para trocar informações entre contextos.

Considerando que o modelo baseia-se em um sistema Multi-Contexto, a integração entre os contextos é feita por meio das regras de ponte que são explanadas no trabalho original. A proposta de *Gelaim, 2016* não especifica questões relacionadas a implementação e operacionalização de agente. Por outro lado, a arquitetura proposta em [Gelaim 2016] é utilizada como base para o trabalho de [Ângelo Gelaim et al. 2018] que, por sua vez, fundamenta o desenvolvimento dos componentes de software mostrados neste artigo.

2.1. Linguagem de Agentes

2.1.1. 3APL

Dastani et al. 2004 [Dastani et al. 2004] apresenta a especificação para uma linguagem de agentes cognitivos. A linguagem é baseada na 3APL (An Abstract Agent Programming Language) e permite ao programador a definição de estados mentais como crenças, objetivos, planos e ações.

As crenças de um agente 3APL descrevem o conhecimento do agente sobre o mundo bem como seu conhecimento acerca do seu estado interno. Elas são especificados e armazenadas na base de crenças do agente. Os objetivos descrevem os estados que o agente deseja alcançar e são especificados pela base de metas do agente. A linguagem de crenças e de objetivos é definida por termos e fórmulas como lógica de primeira ordem. Os termos representam os objetos de domínio e suas fórmulas as relações entre os objetos.

Os planos são responsáveis em atingir os objetivos. Um plano é construído a partir de ações básicas e testes na base de crenças. Planos abstratos são chamados de metas de realização. Uma ação básica especifica os recursos com os quais um agente deve alcançar um determinado estado. O efeito da execução de uma ação básica altera a base de crenças do agente e não o mundo. Por outro lado, ação de teste verifica se uma determinada fórmula é derivável na base de crenças.

Implementar um agente em 3APL significa especificar suas crenças, metas e planos iniciais. Além disso é necessário escrever um conjuntos de regras de revisão de metas, planejar regras de seleção e regras de revisão de planos. A execução é definida por meio de um ciclo de deliberação, que mostra como as regras são aplicadas e em qual ordem devem ser executadas.

Por fim, o trabalho de *Dastani et al. 2004* mostra uma especificação para uma linguagem de agentes, baseada em crenças, objetivos e ações. No trabalho, é definida

toda a relação sintática e semântica dos componentes envolvidos. Porém a definição do ciclo de deliberação, que não faz parte do escopo do trabalho, é fraca. Este fator dificulta a implementação da linguagem e consequentemente a operacionalização de agentes 3APL.

2.1.2. Jason

O trabalho de *Bordini e Hübner, 2006* [Bordini and Hübner 2006] tem como objetivo mostrar as principais funções disponíveis no *Jason*, um framework para o desenvolvimento de agentes BDI baseado em AgentSpeak, uma linguagem de programação orientada a agentes baseada em programação lógica. A arquitetura do AgentSpeak é inspirada em BDI, e em sua versão original é considerada uma linguagem abstrata. Frente a isso, *Bordini e Hübner, 2006* adicionam ao *AgentSpeak* extensões que são necessárias para a execução e operacionalização de agentes.

Jason[1] é implementado em Java, assim como o *framework* proposto neste artigo, as principais funcionalidades disponíveis são: Comunicação entre agentes; anotações em crenças e em planos; execução de sistema multi-agente sobre uma rede; customização em Java das funções do agente; possibilita a implementação de ambiente multi-agente. Considerando que sua implementação é baseada em *AgentSpeak*, o agente é definido com um conjunto de crenças que determinam o estado inicial da base de crenças. A linguagem de crenças é definida como um conjunto de fórmulas atômicas criadas sobre lógica de primeira ordem. Além disso é definido um conjunto de planos que podem satisfazer objetivos. Os objetivos são classificados como *achievement goals* e *test goals*.

Em Jason, um agente pode atuar em ambientes reais ou virtuais. Para facilitar a criação de protótipos é fornecida uma classe *Environment* que pode ser estendida pelo programador para a criação de seus próprios ambientes. Diferente do modelo de agente proposto por [Ângelo Gelaim et al. 2018], em [Bordini and Hübner 2006] os sensores e atuadores ficam atrelados ao ambiente no qual o agente está inserido, essa característica dificulta a integração de agentes *Jason* com ambientes de diferentes topologias e natureza.

3. Proposta

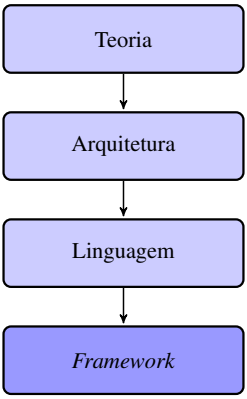
Conforme foi abordado na seção 1, existem diversos desafios, tecnológicos e teóricos, atrelados aos modelos de agentes inteligentes. Frente a isso, esta pesquisa propõe a implementação de um *framework* para permitir a operacionalização de agentes como SMC.

O método de desenvolvimento é adaptado de *Wooldridge e Jennings, 1995*, [Wooldridge and Jennings 1995] que, em seu formato original, divide o processo de desenvolvimento em 3 fases: Teoria; Arquitetura; e Linguagem. Como adaptação ao método, é proposta uma quarta fase denominada *Framework* conforme é mostrado na figura 1.

A teoria, a arquitetura e a linguagem, elementos necessários para a implementação do *framework*, sendo esse o principal objetivo deste trabalho, são definidos na tese *Consciência situacional em agentes autônomos: Raciocínio sobre percepções* de autoria de Thiago Ângelo Gelaim e no artigo *Sigon: A Multi-Context System Language for*

[1]http://jason.sourceforge.net

Figura 1. Método de desenvolvimento de agentes adaptado de *Wooldridge e Jennings, 1995*.



Intelligent Agents de [Ângelo Gelaim et al. 2018] desenvolvido de forma conjunta. Por não ser o objetivo central deste trabalho, os aspectos relativos a teoria, arquitetura e linguagem são apenas superficialmente descritos visando apenas os conceitos utilizados no desenvolvimento do *framework*.

3.1. Teoria

Considerando que o trabalho apresentado em [Ângelo Gelaim et al. 2018] se trata de uma arquitetura genérica, que permite a definição de novos contextos e regras de ponte, é possível representar através dela diferentes tipos de teorias. Por exemplo, a criação de um simples agente reativo ou de uma arquitetura mais complexa envolvendo emoções conforme é mostrado nos trabalhos de [Gelaim et al. 2015, SILVEIRA et al. 2016]. Com o objetivo de corporificar o modelo genérico, este trabalho implementa a teoria do raciocínio prático, por meio de uma arquitetura BDI.

3.2. Arquitetura

A arquitetura do agente define suas lógicas, seus contextos e as regras de ponte. O modelo genérico do agente, conforme definido em [Ângelo Gelaim et al. 2018], é mostrado na Definição 2.

Definição 2 (Modelo de agente proposto)

$$AG = \langle CC \cup \bigcup_{i=1}^{n} C_i, \Delta_{br} \rangle \tag{2}$$

onde CC é o contexto de comunicação e C_i com $1 \leq i \leq n$ os demais contextos do agente e Δ_{br} é o conjunto de regras de ponte; Uma regra br_i conecta dois ou mais contextos [Ângelo Gelaim et al. 2018].

Com base nesse modelo, é definido em [Ângelo Gelaim et al. 2018] uma arquitetura de agente BDI, conforme a Definição 3.

Definição 3 (Definição de um agente BDI)

$$AG_{BDI} = \langle \{CC, BC, DC, IC, PC\}, \Delta_{br} \rangle \quad (3)$$

onde CC, BC, DC, IC, PC são, respectivamente, contextos de comunicação, crenças, desejos, intenções e planejamento, e Δ_{br} é o conjunto de regras de ponte [Ângelo Gelaim et al. 2018].

Além disso, são definidos na arquitetura sensores, atuadores, planos, algoritmo de raciocínio, regras de ponte para uma configuração BDI, entre outros fatores. Sua especificação está na íntegra no trabalho de [Ângelo Gelaim et al. 2018].

3.3. Linguagem

A linguagem é descrita por uma gramática definida em [Ângelo Gelaim et al. 2018]. Utilizando a linguagem foi possível realizar a implementação do *Parser* no framework. A produção principal é definida da seguinte forma:

agent: *communicationContext* (*context* — *bridgeRule*)*;

As demais produções derivadas são mostradas no trabalho de [Ângelo Gelaim et al. 2018].

3.4. Framework

Com o objetivo de permitir a operacionalização da linguagem e a execução do agente, nessa seção é apresentada uma proposta de arcabouço de *software* denominada como *framework*. Sua implementação, realizada em JAVA, é a principal contribuição deste trabalho. Para simplificar o desenvolvimento e desacoplar os componentes, a estrutura a ser implementada neste trabalho segue a especificação de [Ângelo Gelaim et al. 2018], e é dividida em dois componentes principais: *Parser* e *Agent*.

3.4.1. Parser

O componente *Parser* é responsável pela transformação dos arquivos fontes do agente em uma estrutura de dados executável. Para isso ele utiliza a gramática da linguagem apresentada na seção 3.3. A geração do Parser foi feita pela ferramenta Antlr [2] que fornece diversas classes para a representação e tratamento de texto e gramáticas.

3.4.2. Agente

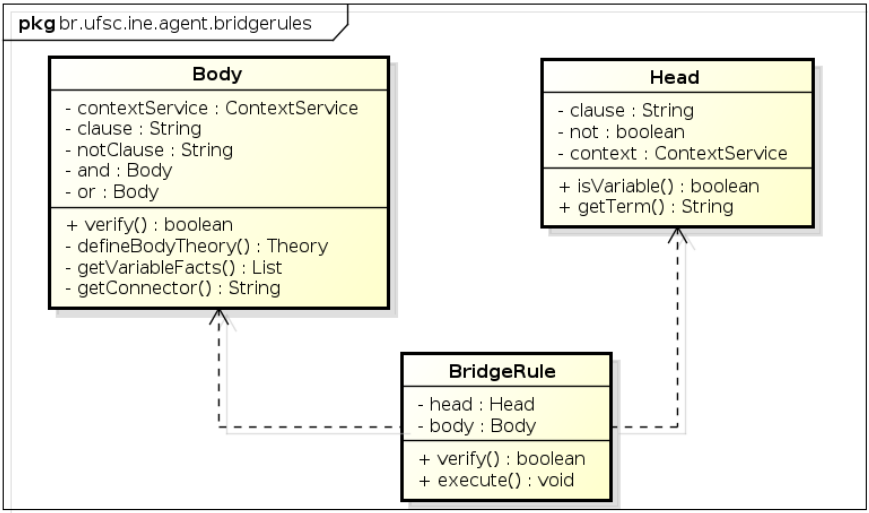
Similar a definição formal de um agente como sistema multi-contexto, a representação do agente como um *software* é feita por decomposição modular usando contextos e regras de pontes. Um contexto, é uma unidade funcional de *software* e uma regra de ponte é utilizada para a troca de informação entre as unidades. Para representar um Agente, é definido no *framework* a classe *Agent* que contém referencias para instâncias de contextos e regras de ponte. Após a execução do *parser* é criada uma instância de Agente e sua execução é inicializada.

[2] http://www.antlr.org/

Um contexto é generalizado no *framework* por meio da *inteface ContextService*, que estabelece métodos para: adição, atualização e busca de teoria, que neste trabalho é representada por uma base de conhecimento; verificação de uma teoria e nome do contexto. Sob a perspectiva da engenharia de *software* um contexto representa um componente que encapsula regras internas e por meio de seus métodos regras de ponte integram conhecimento entre diferentes unidades. A teoria, representada pela classe *Theory*, é implementada na biblioteca tuProlog [3] que fornece um ambiente prolog , sobre o qual é executado o raciocínio utilizando a JVM (*Java-Virtual-Machine*).

Uma regra de ponte tem como objetivo de integrar conhecimento entre os contextos, foi criada uma estrutura de dados para representar as regras de ponte, com base na especificação feita em [Ângelo Gelaim et al. 2018]. A Figura 2 mostra o Diagrama de classes, com os principais métodos, criado para a definição de uma regra de ponte no *framework*.

Figura 2. Diagrama de classe representando uma regra de ponte



Utilizando a estrutura de contextos e regras de pontes, foi criado um agente baseado em uma arquitetura BDI-like, criando contextos de crenças, desejos, intenções, comunicação e planejamento. Além disso, fora criadas regras de ponte para permitir a execução dos ciclos de raciocínio. A definição formal das regras de ponte para representação BDI é feita no trabalho de [Ângelo Gelaim et al. 2018].

[3] http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome

3.4.3. Comunicação com o ambiente

A comunicação do agente com seu ambiente ou com outros agentes é feita pelo contexto de comunicação por meio de sensores e atuadores, não acoplados a uma estrutura de ambiente. Esta característica, permite a operação em ambientes de diferentes topologias. Ou seja, fazendo uso da estrutura dos sensores e atuadores é possível integrar o agente com diferentes tipos de ambientes implementados em diferentes linguagens.

Um sensor deve retornar um valor literal em lógica de primeira ordem ou proposicional. Sua implementação é feita pelo desenvolvedor seguindo a especificação do *framework* que estabelece a classe abstrata *Sensor*. Sob o ponto de vista da arquitetura, um sensor contém um publicador de literais, seguindo o paradigma da programação reativa[4].Considerando que o agente pode ter vários sensores que podem capturar percepções de forma paralela um Sensor é tratado como uma *Thread* implementando a interface *Runnable*.

Para alterar o ambiente ou enviar mensagens para outras entidades, o agente utiliza seus atuadores. Um atuador é definido pelo *framework* como uma classe abstrata *Actuator* com o método *act* que recebe uma lista de argumentos. Considerando que cada ciclo é executada uma única ação diferente de um sensor um atuador não é uma *Thread*.

4. Resultados

Neste seção são apresentados os principais resultados obtidos com a implementação do *framework*. O enfoque da avaliação, alinhado com os objetivos do trabalho, está na execução de um único agente implementado na linguagem. Paralelamente, é realizado um experimento com o uso de contextos e regras de ponte para enfatizar e demonstrar o potencial de expressividade do *framework*.

4.1. Avaliação do caso de uso

Suponha que um Agente é responsável em deslocar em um ambiente que pode conter obstáculos. Caso encontre um obstáculo, o agente deve desviar. O Código 1 mostra a definição deste agente por utilizando a linguagem implementada no *framework*.

Código 1 Representação do agente por meio da linguagem SMC

```
1  communication:
2      sensor("positionSensor", "sensor.PositionSensor").
3      actuator("next", "actuator.NextSlot").
4      actuator("toDodge", "actuator.ToDodge").
5
6  desires:
7          check(slots).
8          not obstacle.
9
10 intentions:
11         end.
12
```

[4]http://reactivex.io/documentation/subject.html

```
13 planner:
14         plan(check(slots),
15              [action(next(slot))], [not obstacle, not end], _).
16
17         plan(check(slots),
18              [action(toDodge(garbage))],
19              [obstacle, not end], not obstacle).
```

Conforme pode ser observado, na linha 1 é criado o contexto de comunicação composto por um sensor (*'positionSensor'*) e dois atuadores (*'next'*, *'toDodge'*). Na linha 6, é criado o contexto de desejos contendo dois termos *'check(slots).'* e *'not obstacle.'* Na linha 10, é criado o contexto de intenções com uma única intenção *'end.'*. Por fim, na linha 13 é criado o contexto de planejamento contendo dois planos definidos nas linhas 14 e 17.

O atuador *'toDodge'* é mostrado no Código 2. Ele especializa a classe *Actuator* que define um atuador no *framework*. Na linha 8, é feita a chamada do método da classe *Main* que, neste exemplo, representa o ambiente. Considerando que o atuador é desacoplado do ambiente, diferente da estratégia utilizada na linguagem Jason, pode-se dizer que a integração com diferentes ambientes é facilitada. Isso também vale para os sensores, que seguem a mesma estratégia de implementação.

Código 2 Implementação de um atuador

```
1  public class ToDodge  extends Actuator {
2
3      public void act(List<String> args) {
4
5          Platform.runLater(new Runnable() {
6              @Override
7              public void run() {
8                  Main.toDodge();
9              }
10         });
11
12     }
13 }
```

O sensor *'positionSensor'* e o atuador *'next'* declarados na definição do agente foram anteriormente detalhados no capítulo 4. Considerando que a implementação do ambiente foge do escopo deste trabalho, a representação da classe *Main* não será detalhada. Por outro lado, é importante destacar que é perfeitamente possível integrar os sensores e atuadores com outras linguagens. Por exemplo, utilizando TCP-IP é possível receber e enviar sinais para um ambiente de realidade virtual implementado em Unity [5] ou com ambientes reais enviando e recebendo sinais de câmeras de trânsito.

Com o objetivo de quantificar o tempo de execução o uso de memória do agente foram feitas dez execuções do Código 3. Na linha 1 é instanciada uma árvore sintática,

[5] https://unity3d.com/pt

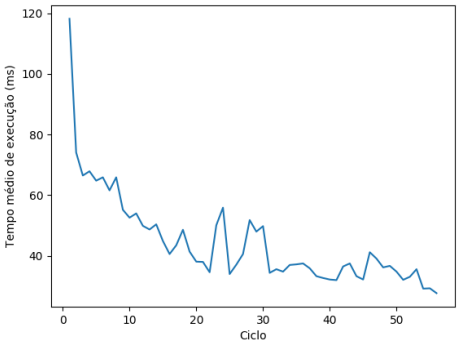
passando como parâmetro o arquivo de definição do agente *"agent.on"*. As linhas 2 e 3 definem a criação de um *walker* para percorrer os nodos da árvore e possibilitar a execução do agente. Na linha 5 é criada uma instancia do agente. Para facilitar a coleta do tempo de execução, de cada ciclo de raciocínio, foi adicionado na linha 6 um arquivo de *profiling*. Por fim, na linha 7 é feita a execução do agente.

Código 3 Código de execução do agente

```
1     ParseTree tree = getParseTree("agent.on");
2     ParseTreeWalker walker = new ParseTreeWalker();
3     walker.walk(new AgentWalker(), tree);
4
5     Agent agent = new Agent();
6     agent.setProfilingFile("esultados.csv");
7     agent.run(agentWalker);
```

Cada uma das dez execuções teve 56 ciclos de raciocínio, o tempo de médio de execução por ciclo é mostrado na Figura 3.

Figura 3. Tempo médio de execução por ciclo



O maior tempo médio de execução foi do primeiro ciclo com 118.2 ms, o menor tempo médio foi do ciclo 56 com 27.7 ms. O tempo médio geral de todos os ciclos foi de 44.46 ms com um desvio padrão de 15.26 ms. Conforme pode ser observado existe um custo inicial alto na execução devido a criação de instancias de objetos JAVA para representar os contextos e regras de ponte. Essa instanciação é realizada uma única vez no primeiro ciclo de raciocínio conforme o agente necessita utilizar os contextos. Além disso a tendência média é o tempo de execução do ciclo diminuir, isso se dá devido as otimizações internas feitas pela JVM após a invocação repetida dos mesmos métodos.

Os testes foram executados em uma máquina com sistema operacional Linux Mint 18 Sarah com 8 GB de memória. A versão da JVM utilizada foi a *openjdk version 1.8.0_162*. O processador utilizado foi o Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz.

Com base nos resultados mostrados, pode-se dizer que a abordagem implementada, representando um agente como Sistema Multi-Contexto, é aceitável considerando os

objetivos do trabalho. Por outro lado, existem pontos a serem melhorados, por exemplo o tempo de custo inicial do agente que está atrelado a criação dos contextos e das regras de ponte. Além disso é necessário realizar testes de estresse com os sensores e atuadores para avaliar o uso do agente em ambientes de muitas percepções. Para maiores detalhes referente a implementação, o exemplo completo está disponível no repositório do Github [6].

4.2. Avaliação de contextos e regras de ponte

Esta seção tem por objetivo avaliar o uso de contextos e de regras de ponte por meio do *framework* implementado. A criação de novos contextos e regras pontes permitem flexibilizar o modelo, adicionando um maior potencial de expressividade. De forma mais específica, usando uma regra de ponte é possível raciocinar sobre diferentes contextos que podem representar qualquer estado mental, por exemplo um contexto emocional.

Para permitir a criação de um contexto qualquer, fora da arquitetura BDI, foi criada a classe *CustomContext* que implementa a interface *ContextService* especificada no capítulo **??**. Para avaliar seu uso, foram criados 3 contextos: *_X, _Z, _Y*, conforme mostra o Código 4.

Código 4 Criação de novos contextos por meio do *framework*.

```
1  CustomContext _x = new CustomContext("X");
2  CustomContext _y = new CustomContext("Y");
3  CustomContext _z = new CustomContext("Z");
4
5  _y.appendFact("a.");
6  _y.appendFact("b.");
7  _y.appendFact("c.");
8
9  _z.appendFact("b.");
10 _z.appendFact("c.");
11 _z.appendFact("d.");
```

O objetivo do experimento é copiar a intersecção dos elementos dos contextos *_Y* e *_Z* para o contexto *_X* que possui um estado inicial vazio. Para isso, foi definida a regra de ponte mostrada no Código 5.

Código 5 Regra de ponte do experimento.

```
1  Head xHead = Head.builder().context(_x).clause("T").build();
2  Body yBody = Body.builder().context(_y).clause("T").build();
3  Body zBody = Body.builder().context(_z).clause("T").build();
4
5  // !_X T :- _Y T & _Z T
6  BridgeRule bridgeRule = BridgeRule.builder().head(xHead)
7      .body(yBody.and(zBody))
8      .build();
```

O *log* da execução do experimento é o seguinte:

[6] https://github.com/valdirluiz/sigon-examples/tree/exemplo-cenario

```
Estado do contexto Y antes execução da regra:
a.
b.
c.

Estado do contexto Z antes execução da regra:
b.
c.
d.

Estado do contexto X após execução da regra de ponte:
b.
c.
```

Conforme pode se observado ao final de execução, após a execução da regra de ponte, o contexto X possui a interseção dos elementos dos contextos Y e Z.

Ao alterar o operador da regra de ponte para *or*, conforme mostra o código 6, o resultado da operação passa a ser a união dos elementos dos contextos *_y* e *_z*.

Código 6 Regra de ponte do experimento.

```
1 Head xHead = Head.builder().context(_x).clause("T").build();
2 Body yBody = Body.builder().context(_y).clause("T").build();
3 Body zBody = Body.builder().context(_z).clause("T").build();
4
5 // !_X T :- _Y T | _Z T
6 BridgeRule bridgeRule = BridgeRule.builder().head(xHead)
7     .body(yBody.or(zBody))
8     .build();
```

O resultado da execução do código 6 é o seguinte:

```
Estado do contexto Y antes execução da regra:
a.
b.
c.

Estado do contexto Z antes execução da regra:
b.
c.
d.

Estado do contexto X após execução da regra de ponte:
a.
b.
c.
d.
```

Apesar do cenário implementado ser simples, por meio desta arquitetura é possível modelar e executar diferentes formas de raciocínio. O código completo do experimento

está disponível no repositório [7] do github.

Por fim, pode-se dizer que os resultados alcançados estão dentro dos objetivos estabelecidos. Por meio do *framework* desenvolvido é possível implementar um agente BDI, fazendo uso de contextos, regras de ponte, sensores e atuadores. Paralelamente, o *framework* também permite a execução de diferentes tipos de raciocínio sem um grande custo de implementação, conforme foi mostrado pelos experimentos com regras de ponte e contextos.

5. Conclusão

Neste trabalho foi apresentada a implementação de um *framework* para a execução de agentes BDI vistos como um Sistema Multi-Contexto. O *framework* implementado seguiu a abordagem de sistemas Multi-Contexto, utilizando como base o modelo proposto por [Ângelo Gelaim et al. 2018]. Foram criadas estruturas de *software* para representar um agente por meio de contextos e regras de ponte. Para permitir a execução do agente, com o auxílio da ferramenta ANTLR [Parr 2013] foi implementado um *parser* descendente recursivo utilizando como base a gramática definida em [Ângelo Gelaim et al. 2018]. No contexto de comunicação, foi criada uma estrutura para a representação e implementação de sensores e atuadores que permitem a integração do agente com diferentes tipos de ambiente ou mesmo com outros agentes.

Por ser baseado em um modelo de agente como SMC, o *framework* permite e favorece a representação de diferentes arquiteturas de agentes. Com o objetivo de demonstrar seu uso, foi proposto um cenário completo de um agente construído sobre uma arquitetura BDI. Paralelamente a isso, foi apresentado um exemplo de uso de regras de ponte e contextos, com o objetivo de mostrar o potencial de expressividade do *framework*.

A literatura apresenta diversos modelos de agente como SMC. Além disso são definidos diferentes arquiteturas de *software* para o desenvolvimento e a operacionalização de agentes, por exemplo JASON e 3APL, ambos mostrados no capítulo **??**. Por outro lado, existe uma deficiência na operacionalização de agentes como SMC, as abordagens apresentadas são restritamente teóricas o que dificulta a realização de testes ou a criação de protótipos. Frente a isso, este trabalho contribui na implementação de um *framework* para a execução e prototipação de agentes baseados em SMC. Com o *framework* desenvolvido se torna possível a operacionalização de agentes modelados baseados em um SMC. A partir desta modelagem, a definição e criação de novas arquiteturas de agente, utilizando contextos e regras de ponte, também é facilitada, aumentando o grau de expressividade do agente.

5.1. Trabalhos futuros

Por se tratar de uma primeira versão de implementação do *framework*, esta pesquisa proporciona várias sugestões de trabalhos futuros. O framework implementado permite a criação e execução de um único agente por vez. A abordagem de sistemas multi agente pode ser melhor explorada em trabalhos futuros.

Apesar da linguagem permitir a criação de novos contextos e novas regras de ponte, fora do escopo BDI e dos contextos de comunicação e planejamento, não é executada nenhuma semântica sobre estes contextos. Uma abordagem futura pode permitir a

[7] https://github.com/valdirluiz/sigon-examples/blob/tcc-valdir/src/avaliacao/contextos/AvaliacaoContextos.java

integração destes novos contextos e regras de ponte com o modelo BDI já implementado por padrão no *framework*.

Além disso, existem aspectos a serem melhorados na implementação do *parser*. Na versão atual não é feita nenhuma validação semântica sobre o código fonte do agente, essas validações podem ser tratadas e analisadas em trabalhos futuros. Também é necessário executar testes de stress envolvendo os atuadores e sensores para verificar o comportamento do agente em ambientes muito dinâmicos.

Por fim, esta pesquisa focou no desenvolvimento de um *framework* para a execução de agentes usando representações de conhecimento baseadas em lógica de primeira ordem e proposicional. A integração e criação de contextos com diferentes tipos de raciocínio, por exemplo com ontologias, pode ser implementada em trabalhos futuros.

Referências

Bordini, R. H. and Hübner, J. F. (2006). Bdi agent programming in agentspeak using jason. In Toni, F. and Torroni, P., editors, *Computational Logic in Multi-Agent Systems*, pages 143–164, Berlin, Heidelberg. Springer Berlin Heidelberg.

Casali, A. (2008). *On intentional and social agents with graded attitudes*. PhD thesis, Universitat de Girona.

Dastani, M., van Riemsdijk, M. B., Dignum, F., and Meyer, J.-J. C. (2004). A programming language for cognitive agents goal directed 3apl. In Dastani, M. M., Dix, J., and El Fallah-Seghrouchni, A., editors, *Programming Multi-Agent Systems*, pages 111–130, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gelaim, T. Â. (2016). Modelo de agentes e-BDI integrando confiança baseado em sistemas multi-contexto. Master's thesis, Universidade Federal de Santa Catarina.

Gelaim, T. , Silveira, R. A., and Marchi, J. (2015). Towards a model of cognitive agents: Integrating emotion on trust. In *2015 Fourteenth Mexican International Conference on Artificial Intelligence (MICAI)*, pages 80–86.

Herzig, A., Lorini, E., Perrussel, L., and Xiao, Z. (2017). Bdi logics for bdi architectures: Old problems, new perspectives. *KI - Künstliche Intelligenz*, 31(1):73–83.

Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

Rao, A. S. and Georgeff, M. P. (1995). Bdi agents: From theory to practice. In *ICMAS*, pages 312–319.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. 3rd edition.

Sabater, J., Sierra, C., Parsons, S., and Jennings, N. R. (2000). Using multi-context systems to engineer executable agents. In Jennings, N. R. and Lespérance, Y., editors, *Intelligent Agents VI. Agent Theories, Architectures, and Languages*, pages 260–276, Berlin, Heidelberg. Springer Berlin Heidelberg.

SILVEIRA, R., BITENCOURT, G. K. D. S., Ângelo GELAIM, T., MARCHI, J., and PRIETA, F. D. L. (2016). Towards a model of open and reliable cognitive multiagent systems: Dealing with trust and emotions. *ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal*, 4(3).

Wooldridge, M. (2002). *Intelligent Agents: The Key Concepts*, pages 3–43. Springer Berlin Heidelberg, Berlin, Heidelberg.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152.

Ângelo Gelaim, T., Hofer, V. L., Marchi, J., and Silveira, R. A. (2018). Sigon: A multi-context system framework for intelligent agents. *Expert Systems with Applications*.

APÊNDICE A – Código Fonte dos experimentos

A.1 CÓDIGO FONTE DO AGENTE DESENVOLVIDO

Implementação do sensor *PositionSensor* :

```
public class PositionSensor extends Sensor {

    public static final PublishSubject<String>
    positionObservable = PublishSubject.create();


    public void run() {
        positionObservable.subscribe(super.publisher);
    }
}
```

Implementação do atuador *NextSlot* :

```
public class NextSlot  extends Actuator {

    public void act(List<String> list) {
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                Main.nextSlot();
            }
        });

    }
}
```

Implementação do atuador *ToDodge* :

```
public class ToDodge  extends Actuator {

    public void act(List<String> list) {

        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                Main.toDodge();
            }
        });
    }
}
```

Código fonte para a execução do agente:

```
File agentFile = new File("r1.on");
CharStream stream = CharStreams.fromFileName(agentFile.getAbsolutePath());
AgentLexer lexer = new AgentLexer(stream);
CommonTokenStream tokens = new CommonTokenStream(lexer);

AgentParser parser = new AgentParser(tokens);
parser.removeErrorListeners();
parser.addErrorListener(new VerboseListener());

ParseTree tree = parser.agent();
ParseTreeWalker walker = new ParseTreeWalker();

AgentWalker agentWalker = new AgentWalker();
walker.walk(agentWalker, tree);

Agent agent = new Agent();
agent.setProfilingFile("/home/valdirluiz/resultados.csv");
agent.run(agentWalker);
```

Código fonte para a execução do agente:

```
communication:
    sensor("positionSensor", "sensor.PositionSensor").
    actuator("next", "actuator.NextSlot").
    actuator("toDodge", "actuator.ToDodge").

desires:
        check(slots).
        not obstacle.

intentions:
        end.

planner:
        plan(check(slots), [action(next(slot))], [not obstacle, not
        end], _).
        plan(check(slots), [action(toDodge(garbage))], [obstacle, not
        end], not obstacle).
```

A.2 EXPERIMENTO COM REGRAS DE PONTE

Regra de ponte para representação de intersecção de conjuntos:

```
CustomContext _x = new CustomContext("X");
CustomContext _y = new CustomContext("Y");
CustomContext _z = new CustomContext("Z");

_y.appendFact("a.");
_y.appendFact("b.");
_y.appendFact("c.");

_z.appendFact("b.");
_z.appendFact("c.");
_z.appendFact("d.");

Head xHead = Head.builder().context(_x).clause("T").build();
Body yBody = Body.builder().context(_y).clause("T").build();
Body zBody = Body.builder().context(_z).clause("T").build();

BridgeRule bridgeRule = BridgeRule.builder()
.head(xHead)
.body(yBody
.and(zBody)).build();

bridgeRule.execute();

println("Estado do contexto Y antes execução da regra: ");
println(_y.getTheory().toString());

println("Estado do contexto Z antes execução da regra: ");
println(_z.getTheory().toString());

println("Estado do contexto X após execução da regra de ponte: ");
println(_x.getTheory().toString());
```

Regra de ponte para representação de união de conjuntos:

```
CustomContext _x = new CustomContext("X");
CustomContext _y = new CustomContext("Y");
CustomContext _z = new CustomContext("Z");

_y.appendFact("a.");
_y.appendFact("b.");
_y.appendFact("c.");

_z.appendFact("b.");
_z.appendFact("c.");
_z.appendFact("d.");

Head xHead = Head.builder().context(_x).clause("T").build();
Body yBody = Body.builder().context(_y).clause("T").build();
Body zBody = Body.builder().context(_z).clause("T").build();

BridgeRule bridgeRule = BridgeRule.builder()
.head(xHead)
.body(yBody
.or(zBody)).build();

bridgeRule.execute();

println("Estado do contexto Y antes execução da regra: ");
println(_y.getTheory().toString());

println("Estado do contexto Z antes execução da regra: ");
println(_z.getTheory().toString());

println("Estado do contexto X após execução da regra de ponte: ");
println(_x.getTheory().toString());
```

ANEXO A – Gramática da Linguagem

Gramática da linguagem definida por *GELAIM et al.* (GELAIM et al., 2018):

```
grammar Agent;

agent
        :
        communicationContext (context | bridgeRule)*
        EOF
        ;

context
        :
        logicalContext | functionalContext
        ;



bridgeRule
        :
        head ':-' body '.'
        ;

logicalContext
        :
        logicalContextName ':' formulas
        ;

functionalContext
        :
        communicationContext |
        plannerContext
        ;

plannerContext
        :
        PLANNER  ':' plansFormulas
        ;
communicationContext:
        COMMUNICATION ':' (sensor | actuator)+
        ;

logicalContextName
        : primitiveContextName
        | customContextName
;
```

```
primitiveContextName
        : (BELIEFS | DESIRES | INTENTIONS)
        ;

customContextName
        :
        '_'(LCLETTER | UCLETTER)+ character*
        ;


plan
        : PLAN '(' somethingToBeTrue ','
        compoundAction
        (',' planPreconditions ',' internalOperator? planPostconditions)?
        (',' cost)? ')'
        ;



somethingToBeTrue
        : listOfClauses
        ;

planPreconditions
        : conditions
        ;

planPostconditions
        : conditions
        ;

conditions
        : ('_' | listOfClauses)
        ;


action
        : ACTION '(' functionInvocation (','
        actionPreconditions ',' internalOperator?
        actionPostconditions)? (',' cost)? ')'
        ;

actionPreconditions
        : conditions
```

```
        ;

actionPostconditions
        : conditions
        ;

functionInvocation
        : functionName '(' argumentList? ')'
        ;

functionName
        : LCLETTER + character*
        ;

sensor
    : SENSOR '('  sensorIdentifier  ',' sensorImplementation ')' '.'
    ;


sensorIdentifier
    : STRING
    ;

sensorImplementation
    : STRING
    ;

actuator
    : ACTUATOR '(' actuatorIdentifier ','
    actuatorImplementation ')' '.'
    ;


actuatorIdentifier
    : STRING
    ;

actuatorImplementation
    : STRING
    ;

internalOperator
        : beliefAdition
        | beliefRemotion
        | desireAdition
```

```
        | desireAdition
        ;

beliefAdition
        : '+'
        ;
beliefRemotion
        : '-'
        ;

desireAdition
        : '+!'
        ;
desireRemotion
        : '-!'
        ;
argumentList
        :       expression (',' expression)*
;

expression
        : constant | variable
        ;

compoundAction
        : ('[' action (',' action)* ']') |'_' ;

clause
        : propClause | folClause
        ;

listOfClauses
        : clause
        | ('[' clause (',' clause)* ']')
        ;

formulas
        : (propFormula        | folFormula )*
        ;

propFormula
        : (propClause  (':-' propLogExpr )?)'.'
        ;

folFormula
```

```
        : (folClause ( ':-' folLogExpr )?) '.'
        ;

plansFormulas
        : ((plan  | action )'.') *
        ;


contextName:
        logicalContextName | PLANNER | COMMUNICATION
        ;


head
        :
('!' negation?  contextName )  (clause | negation? variable)
;

body
        : negation? contextName   ((clause | negation? variable) | plan)
((AND | OR) negation?  contextName
((clause | negation? variable) | plan))*
        ;


propClause
        : negation? constant (annotation)?
        ;

folClause
        : negation? constant '(' (term) (',' (term) )* ')' (annotation)?
        ;

term
    : (numeral | constant | variable | '_')
    (operator (numeral | constant | variable | '_') )?
    ;

operator
    : '<' | '=<' | '>' | '>=' | '-' | '+'
    ;

negation
        : 'not' | '~';
```

```
annotation
     : (preAction| gradedValue)
     ;

preAction
    : '['constant']'
    ;

gradedValue
    : '->0.' numeral
    ;
cost
    : '0.' numeral
    ;

numeral
        : DIGIT+
        ;

constant
        : LCLETTER character*
        ;

variable
        : UCLETTER character*
        ;

propLogExpr
        :  propClause propLogExprL
        ;

propLogExprL
        : (AND | OR) propClause propLogExprL |
        ;


folLogExpr
        :  folClause folLogExprL
        ;
folLogExprL
        : (AND | OR) folClause folLogExprL |
        ;


character
```

```
    : LCLETTER | UCLETTER | DIGIT
    ;

/*
* TODO: user been able to add a semantic for a context.
*semanticRules
*        : (LCLETTER | UCLETTER) character* '.semantic'
*        ;
*/

AND
   : '&'
   ;

OR
   : '|'
   ;


STRING
        :
    '"' (~["\\\r\n] | '\"')* '"';



BELIEFS
        : 'beliefs'
        ;

DESIRES
        : 'desires'
        ;

INTENTIONS
        : 'intentions'
        ;
PLANNER
        : 'planner'
        ;

COMMUNICATION
        : 'communication'
        ;
SENSOR
        : 'sensor'
```

```
        ;
ACTUATOR
        : 'actuator'
        ;
PLAN
        : 'plan'
        ;
ACTION
        : 'action'
        ;


LCLETTER
    : [a-z_];

UCLETTER
    : [A-Z];

DIGIT
    : [0-9];

WS
   : [ \t\r\n] -> skip
;


BlockComment
    :   '/*' .* '*/'
        -> skip
    ;

LineComment
    :   '//' ~[\r\n]*
        -> skip
;
```